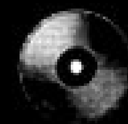
PPS パーソナルプログラミングシリーズ

CD-ROM INCLUDED 

Visual C++6.0 プログラミング入門

桜田 幸嗣／田口 景介 著

koji Sakurada and keisuke taguchi

PPS Series

アスキー出版局

Visual C++6.0
プログラミング入門

桜田 幸嗣／田口 景介 共著

アスキー出版局

商 標

※本書は、一般書店で販売されている「Visual C++ 6.0 プログラミング入門」（弊社刊）と同内容の本を Visual C++ 6.0 Standard Edition に添付したものです。

はじめに

世の中にはどうやら２種類の人間が存在するようです。

ある人々はとにかく目的地を目指し、途中の道筋にはあまりこだわりません。終わりよければすべてよし。ロールプレイングゲームで遊ぶとき、洞窟の宝物はいくつも取り忘れるけれど、出口の光を見るのは誰よりも早いというタイプです。

ところがその一方で、周囲を確かめながら一歩一歩進むのが好きな人々もいます。このタイプの人たちは「なんだかよくわかんないけど、できちゃったー！」という場当たり的な結果を嫌います。ですから洞窟に入ったら、すべての経路をしらみつぶしに踏破しなくては気がすみません。そのせいで罠に引っかかったりもするわけですが…

どちらが良い悪いの問題ではありません。「人間には（一人の人間の心の中にだって）そういう２つの性向が混在しているものだ。」と、皆さんにも思いあたるところがおありでしょう？

………

さてここで、Windowsプログラミングの世界へと目を転じてみます。ご存じのように、今のところWindowsプログラマには、２つのメジャーなプログラミング手段が提供されています。つまりVisual BasicとVisual C＋＋の２つです。

Visual BasicとVisual C＋＋は、それぞれプログラミング言語として異なる特徴を持ち、それぞれにふさわしい用途があります。といっても、一昔前によくいわれていた「アマチュア用のBASIC、プロ用のC」などという構図とはちょっとばかり違います。そう、実は前述の人間の行動様式の違いというものが、ここに深く関わってくるのです。

Visual Basicは、何はともあれ結果を早く得たい、要するに「プログラムは正しく動けばよいのだ」（これも立派な１つの見識です）という立場のプログラマにお勧めしたいツールです。Visual Basicはあなたの最終目標に向けて、まっすぐな道を引いてくれるでしょう。

それに対してVisual C＋＋によるプログラミングは、Windowsのダンジョンを自力でガンガン進むようなものです。なるほど、先へ進むには多少のパワーは必要です。袋小路に突き当たることだってあります。でもその見返りとして、自分の好きな場所にどこにでも足を踏みいれることができます。そして、Windowsに関する知識という「宝物」が山ほど手に入ります。

繰り返しますが、どちらがエライというのでもありません。Visual Basicによるスマートなプログラミングと、Visual C＋＋によるパワープログラミング、この両方をこなせるならば、それに優るものはないでしょう。ただし、筆者らの体験に照らしてみて、こういうことはいえると思います。Visual Basicの世界しか知らないプログラマは、Windowsプ

ログラミングの本当のおもしろさをまだ知らない！ これは絶対に損ですぜ、皆さん。

………

さて「Visual C＋＋プログラミングはダンジョンだ」なんて文字を見て、皆さんがあまりおビビりにならないうちに、ひとこと付け加えておきましょう。安心してください。Visual C＋＋は、これ自体が Windows プログラミングの強力な助っ人なのです。

かつて、C 言語を使った Windows プログラミングは、確かに非常に手間のかかるものでした。しかし Visual C＋＋を使えば、面倒な作業は付属のツールがサポートしてくれますし、定型処理部分の（ほぼ）自動的なコーディングさえしてくれるので、プログラマはポイントとなる部分だけに注意を集中することができます。また Windows のために設計されたライブラリのおかげで、GUI を利用した独特のスタイルのプログラムが、簡単に、かつ簡潔に記述できます。

本書が目的とするのは、Visual C＋＋に用意されたこれらのツールやライブラリの活用方法を体験してもらい、そこから生まれる知的な満足感の一端を味わってもらうことです。決して Windows プログラミングのすべてを網羅しようとはしていません。

なんだかんだいっても、Windows の独特なプログラミングスタイル、膨大なライブラリなど、Visual C＋＋を志すプログラマの前には高い壁があります。何も手掛かりのない状態では、どこから手をつけていいものか見当がつかないことも多いでしょう。まだ右も左もわからない、そんな方々のために、本書はとりあえず「前を向く方法」をお伝えしようと思います。まずは足元を固めて、膨大な情報の中から重要なものとそうでないものを見分ける力を養えば、あとは独力で新しいプログラミングの世界を進んでいくことができるはずです。本書がその確実な第一歩への助けになれば幸いです。

………

Visual C＋＋ 1.0 に合わせて出版された「Visual C＋＋プログラミング入門」ですが、Visual C＋＋のバージョンアップと同期して 3 回の改訂を経ることになりました。Visual C＋＋ 6.0 では主に Internet Explorer 4.0 で追加されたコントロールのサポートが行われ、また引き続き大幅な統合環境の改善が行われています。本書では第 4 部において、Internet Explorer 4.0 とともに利用できるようになった HTML 表示機能を使って、簡単な WWW ブラウザを作成しています。また Appendix における言語解説を拡張し、要望の多かった C++言語でのポインタの扱い方の解説を加えました。

本書を執筆するにあたって、たくさんの方々にお世話になりました。とくにいくつもの有益な助言をくださった遠藤孝信氏、松崎武志氏、西村克信氏、鈴木博和氏、田畑康彦氏、そして編集担当の川崎晋二氏に感謝いたします。

筆者一同

本書の構成

本書は、Microsoft Visual C++ Development System Version 6.0(以下 Visual C++)を利用した Windows アプリケーションの作成方法を詳しく解説した Windows プログラミングの入門書です。

Visual C++は、Windows が提供するグラフィカルユーザーインターフェイスを利用して効率的にプログラムを開発できる、非常に優れたプログラム開発環境です。本書は、C プログラマを対象に、さまざまなツールの利用法、Windows プログラミングの基礎知識、ユーザーとの対話、ファイル入出力など、Windows 上で動作するプログラムを作成するのに欠かせない知識について詳しく解説したものです。

● 対象とする読者

本書では、構造体、変数、データ型などについては基本的に既知のものとして、本文ではとくに説明をしていません。したがって、本書の内容を十分に理解するには C 言語に関する知識が必要です。C++言語に関しては、本文中のコラムや Appendix である程度の説明は行いますが、それ以上に C++言語について知りたいという場合には Visual C++に付属のチュートリアル、市販の参考書などを参照してください。

本書を読み進めていくのに、C++言語に関する知識、Windows プログラミングの経験などはとくに必要ありませんが、これらの知識や経験があれば本書の内容を理解するのに大きな助けとなることでしょう。

● 本文の構成

本書は全 4 部で構成されており、その内容は以下のようになっています。

第 1 部　Visual C++に触ってみよう

第 1 部では、Visual C++が提供するプログラム開発環境をはじめとして、Visual C++を利用したプログラミングの流れ、作成したプログラムの構造、プログラムを起動したあとの実行の流れなど、Visual C++プログラミングに必要な基礎知識について説明をしています。また、C++言語に関する基礎知識、Visual C++と Windows プログラミングとの関係などについても第 1 部で簡単に説明を行います。ここから、Visual C++プログラミングの世界が広がっていきます。

第2部　Visual C++プログラミングの基本を押さえよう

第2部では、いくつかのサンプルプログラムを通して、GDI、メニュー、ダイアログボックス、マウス／キーボード入力といった Windows のユーザーインターフェイスを介してユーザーと対話をする方法について説明をします。同時に Visual C++が提供する AppWizard、リソースエディタ、ClassWizard などのツールの実際的な利用方法についても説明します。また、プログラム作成時に欠かすことのできない作業――デバッグ――についても解説をします。これらの知識は Visual C++プログラミングを進めていく上で重要な基礎体力となるはずです。

第3部　MFC を使ってみよう

第3部では、テキストエディタ+ドローツールを兼ね備えたプログラムを作成しながら、MFC を使用して本格的なプログラムを作成するために必要な知識について説明をします。Visual C++が提供する機能を利用したファイル入出力の方法や、リストを使ったデータの管理などは、本格的なプログラムを作成するときには必須の知識です。

第4部　Windows らしいプログラムを作ってみよう

第4部では、簡単な WWW ブラウザを作成しながら、最近 Windows に導入された2つのコントロール（ツリービューコントロールと HTML ビューコントロール）の使い方を説明していきます。コントロールを細かく制御した高度なアプリケーションを作成するのにこれらの知識が役立つはずです。

Appendix

Appendix では、本書の内容を理解するのに必要な C++言語およびポインタ変数に関する解説と、Visual C++ 6.0 よりサポートされたドキュメント－ビュー・アーキテクチャを使用しない MFC アプリケーションについての解説を掲載しています。C++言語についてより深く知りたいという方は、Visual C++に付属のチュートリアル、市販の C++言語の解説書などを参照してください。

● 付属 CD-ROM

本書で作成するプログラムのソースコードはすべて付属の CD-ROM に収めてあります。これらのプログラムをコンパイル／実行するには Visual C++ 6.0 の処理系と Visual C++ 6.0 が動作する環境が必要です（実行可能ファイルは収めていません）。付属 CD-ROM の利用法については、このあとの「付属 CD-ROM について」を参照してください。

付属CD－ROMについて

本書で作成するプログラムはすべて、付属のCD-ROMに収めてあります。これらのプログラムをコンパイル／実行するには、Visual C++ 6.0の処理系とVisual C++ 6.0が動作する環境が必要です。

● 付属CD-ROMの構成

付属CD-ROMには、以下に示すようなフォルダ構成でプログラムが収められています。

フォルダ	内容	扱っている章
URLMan	URLマネージャ	第4部1～4章
bug	デバッグ用サンプルプログラム	第2部5章
DlgTest	ダイアログボックスのテスト	第2部3章
G1	ペンのテスト	第2部1章
G2	ブラシのテスト	第2部1章
G3	ビットマップ転送のテスト	第2部1章
Hello	画面に「Hello World!」と表示	第1部2章
MenuTest	メニューのテスト	第2部2章
MMView	テキストエディタとドローツール	第3部1～2章
Paste	マウスとキーボード入力の受け付け	第2部4章

さらに「URLMan」フォルダと「MMView」フォルダの下には以下のようなサブフォルダがあります。

「URLMan」フォルダ	内容
URLMan	本書で解説をしたプログラムの実装
Resource	URLManで使用するリソースを抽出したフォルダ
IsURL	URLManで使用するデータファイルを作成するためのユーティリティ

「MMView」フォルダ	内容
Complete	すべての実装を行ってあるもの
Step0	スケルトン
Step1	テキストエディタの実装が済んだもの
Step2	ドローツールの実装が済んだもの（その 1）
Step3	ドローツールの実装が済んだもの（その 2）

● 付属 CD-ROMファイルのコピー

以上のファイルは、付属 CD-ROM に無圧縮状態で収められています。これをコンパイル／実行する場合には、ハードディスク上の適当な位置にコピー用のフォルダを作成し、そこにコピーすることをお勧めします。また、付属 CD-ROM からハードディスクにファイルをコピーした場合、それらのファイルには読み取り専用属性が付きますので、プロパティシートを使用したり、DOS プロンプトで attrib コマンドを使用するなどして、読み取り専用属性をはずす必要があるので注意してください。

各プログラムの詳しい内容については、本文中の説明を参照してください。付属フロッピーディスクに掲載したすべてのプログラムは、以下の環境でのコンパイル／実行を確認しました。

CPU	MICRON MILLENNIA (Pentium 166MHz)
メモリ	32M バイト
ハードディスク	1G バイト
OS	Windows 98
処理系	Microsoft Visual C++ 6.0 Standard Edition

● 注意

本書および付属 CD-ROM に収められているプログラムの著作権は著作者に、出版権は株式会社アスキーにあります。これらのプログラムは私的利用の範囲での使用、流用、変更、複製についてはこれを認めます。ただし、本書および付属 CD-ROM に掲載されているプログラムの一部を流用、または変更したプログラムの運用結果については、著作者、および株式会社アスキーは、いっさい責任を負いかねますのでご了承ください。

目次

第2部 Visual C++プログラミングの基本を押さえよう

第3部 MFCを使ってみよう

第4部 Windowsらしいアプリケーションを作ってみよう

Appendix

第1部

Visual C++に触ってみよう

Visual C++プログラミングという新世界への第一歩を踏み出すときが来ました。第1部では、Visual C++を使ったWindowsプログラミングがどんなものかを理解するために、簡単なプログラムを作成します。実際にコードを書くのは、第1部を通してほんの数行程度にすぎないのですが、この数行の意味を理解するまでの道のりは決して平坦なものではありません。とはいえ、Visual C++の動作原理を知らないままハイレベルな内容に手を付けるのも、生兵法となりかねません。この段階ではまだ先を急がず、疑問に感じたことを1つ1つクリアしていく姿勢が重要です。まずは基本をしっかり押さえましょう。

1　Visual C＋＋とは？

Visual C＋＋は単なるC＋＋コンパイラではありません。これはWindowsアプリケーションを作成するための統合環境です。開発を支援する数々のツール、Windowsの能力を引き出すライブラリ、その他いろいろな便利なパーツが密接に結び付いたシステム、それがVisual C＋＋です。しかし構成要素が多いだけに、Visual C＋＋の全体像は、なかなか見通すことができません。そこでまず、本章ではVisual C＋＋の世界を作り上げているパーツの役割を１つ１つ見ていきます。Visual C＋＋という開発環境が登場してきた背景と、そのおおまかな構成をここで頭に入れてください。

1.1　GUI時代の開発環境

Windows 95の爆発的な流行とともに、日本でもグラフィック画面をベースとするパソコン操作環境が一般のものとして定着しました。いわゆる**GUI（Graphical User Interface）**というやつです。GUIは操作方法が直感的でわかりやすく、しかも見た目がおしゃれです。アプリケーションユーザーから見て、こんな結構なものはありません。

しかしその反面、GUIの登場によって、アプリケーションの作成はとても面倒で手間のかかる作業となりました。たとえば画面に"Hello"と表示するプログラムを考えてみましょう。Visual C++が登場する以前には、これほど単純なことですら、Windowsで実現しようと思ったら、「ウィンドウを作って位置を決めて表示して……」と何十行ものコードを必要としていたのです。しかも、そのコードの大部分は本来の目的（文字列表示）にはほとんど関係がありません。

Visual C＋＋以前のWindowsプログラミングでは、GUI採用のために増加したこのような作業は、すべてプログラマの負担となっていました。Windowsプログラミングに手を染めるプログラマには、大いなる根性と体力が必要だったのです。

そんな状況の中で登場したのがVisual C＋＋ 1.0でした。Visual C＋＋ 1.0を使って

“Hello”を表示するプログラムを作れば、プログラマが書かねばならないコードはわずか1行、しかもそれは文字列を画面に表示するコードそのものです。それ以外の部分はみんなVisual C++が面倒を見てくれます。すなわち、Visual C++を利用することによって、プログラマは煩わしい作業から解放され、本来の目的のみに集中できるようになるのです。そして今、Visual C++はメジャーバージョンが6となり、プログラマをさらに強力に支援してくれる開発環境となったのです。

プログラマが簡単に効率的にプログラム作成を行えるようにするために、Visual C++は多くの便利なツールやライブラリを提供してくれます。次節では、プログラマを手助けしてくれるこれらのさまざまなツール類について説明をすることにします。

1.2 開発環境のスタンダード、Developer Studio

Visual C++を起動すると、**図1-1**に示すようなウィンドウが画面に表示されます。このウィンドウがこれからのWindowsプログラム開発環境のスタンダード、Developer Studioです。

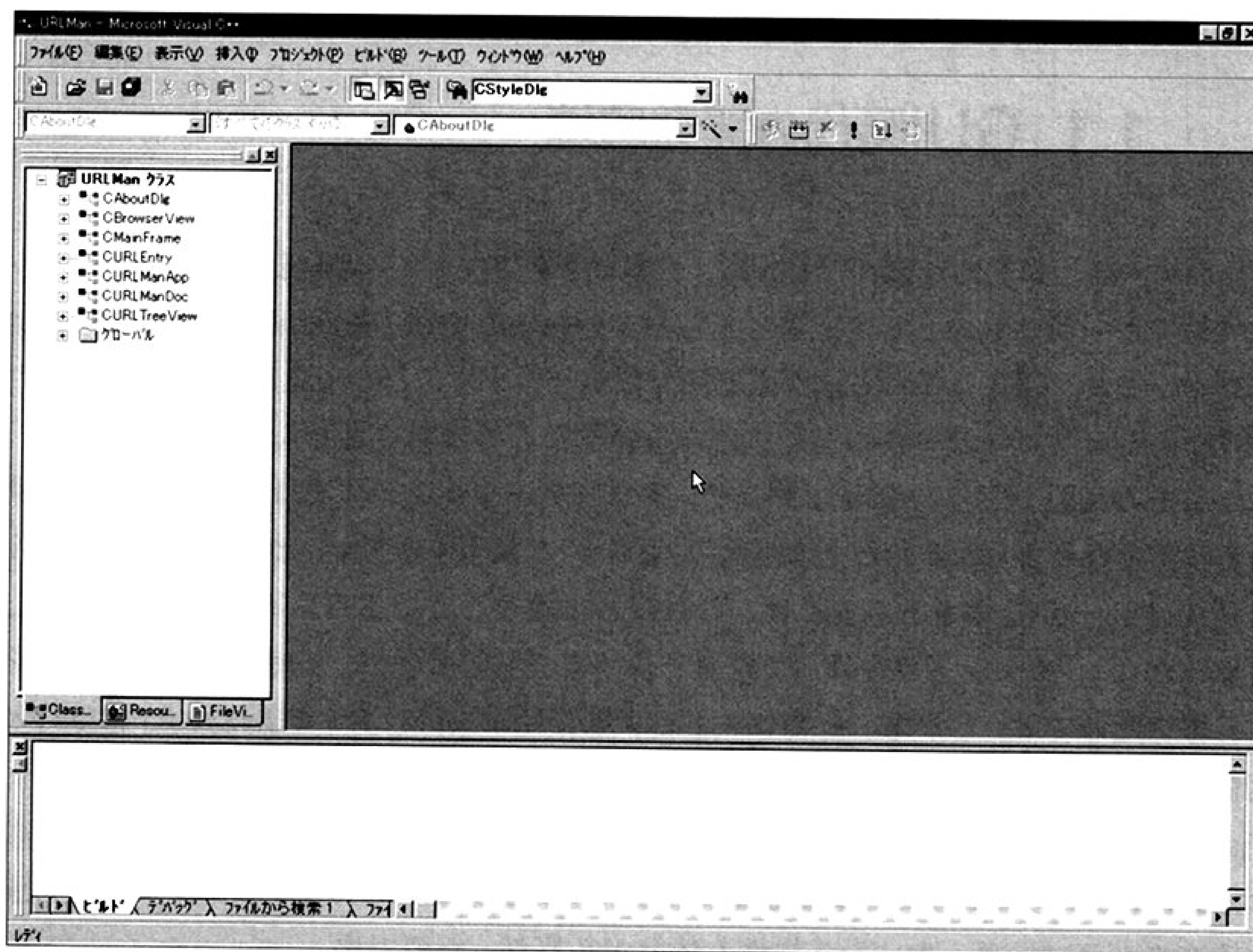


図1-1 Developer Studio

ひとことでいうと、Developer StudioはVisual C++の作業全体のまとめ役です。Developer Studioでは、プロジェクトの新規作成に始まって、ソースコードやリソースの編集、コンパイル、実行、デバッグなど、Windowsプログラム開発を1つの環境の中で行うことができるのです。

では、Visual C++とDeveloper Studioがどのような構成で、どのような機能を持っているのか、少し覗いてみることにしましょう。

● Developer Studioの画面構成

図1-1に示したように、Developer Studioは、ExcelやWordといった標準的なWindowsアプリケーションと同様に、タイトルバー、メニュー、ツールバー、ステータスバーといった部分から構成されていますが、そのほかにもプログラム開発を快適に行うためにいくつかの要素を備えています。

その1つが、**ワークスペースウィンドウ**です。ここには、現在作業を行っているプロジェクト(プログラム)を構成する、ファイルやクラス、およびリソースをページに分けて表示することができます。また、その右側のスペースではプログラムコードやリソースの編集を行います(**図1-2**)。コンパイル時やデバッグ時にはDeveloper Studioの下部に専用のウィンドウが表示され、さまざまな情報をプログラマに示してくれます。

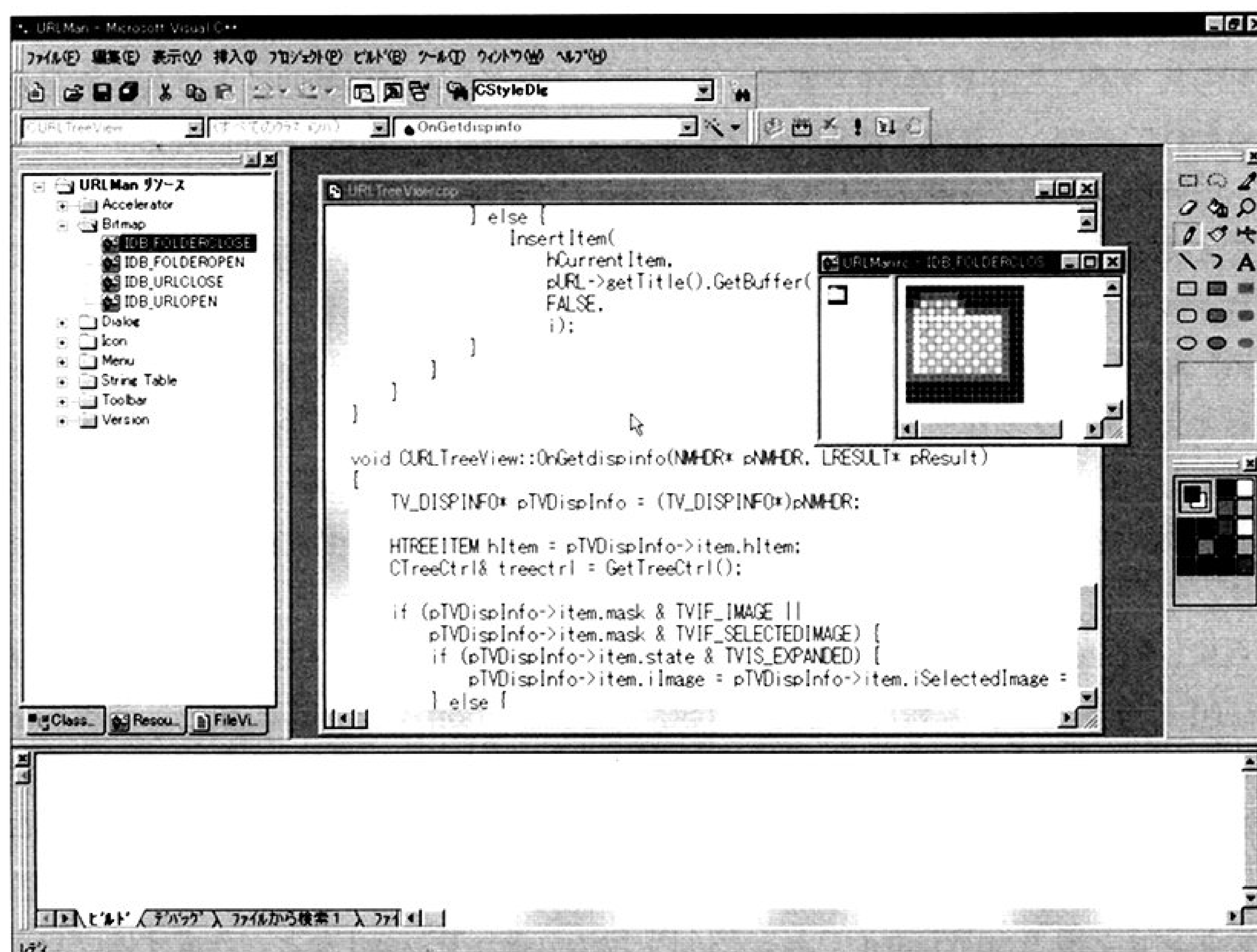


図1-2 Developer Studioで作業中の場面

● MFC AppWizardとカスタム AppWizard

プログラマがVisual C++から受けるご利益はいろいろありますが、なかでも特筆すべき存在が、**MFC AppWizard**というツールです（**図1-3**）。なにしろこのMFC AppWizardは、人間様に代わってプログラムを作ってしまうのです。

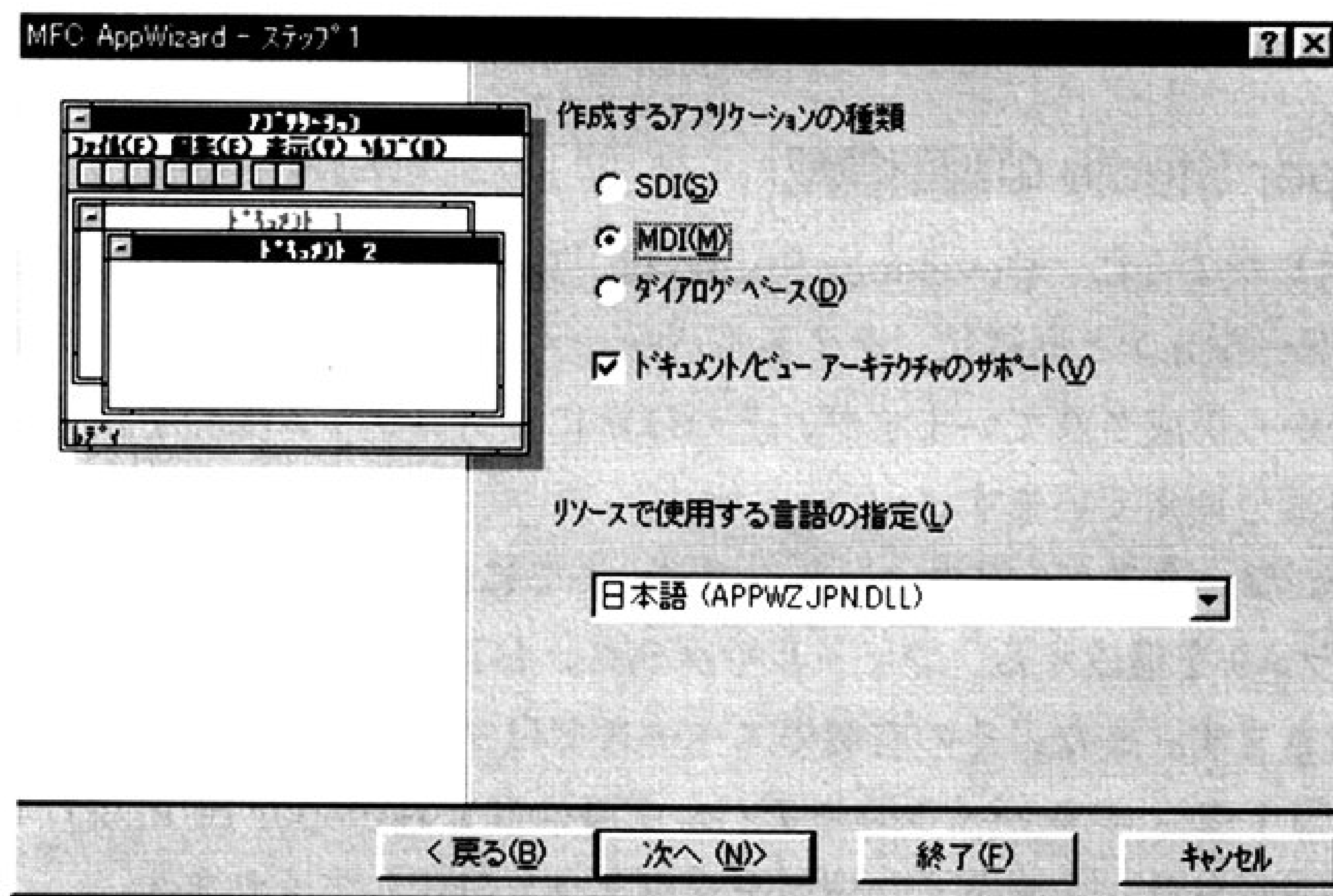


図1-3 MFC AppWizard

もちろんMFC AppWizardを使っても完璧なプログラムが自動的にドカスカ大量生産されるわけではなく、作成されるのは**スケルトン**（Skeleton：骨格）と呼ばれるアプリケーションの枠組みにすぎません。しかし、Windowsアプリケーションは、この枠組みを作り上げるまでが一仕事なので、この部分をMFC AppWizardが受け持ってくれることによりプログラミングの手間は大幅に軽減することになります。

MFC AppWizardが作り出すスケルトンは、なかなかあなどれない底力も秘めています。たとえば「アプリケーションにツールバーを付けたい！」、「テキストデータをプリントアウトしたい！」、という程度の機能なら、スケルトンの中で実現してくれます。自分でコードを書く必要なんてどこにもありません。すべてMFC AppWizardが行ってくれるのです。

さらに、Professional Edition以上のバージョンのVisual C++ 6.0では、**カスタムAppWizard**というツールも利用できます（Standard Editionでは、この機能はサポートされない）。カスタムAppWizardを使えば、通常のMFC AppWizardで作成するスケルトンをさらにカスタマイズしたり、まったく独自のスケルトンを作成できます。そのほかにもActiveXコントロール用AppWizardであるMFC ActiveX ControlWizard、IISエクステンション用AppWizardであるISAPI Extension Wizardなど、作成するプログラムのタイプによって複数のAppWizardが用意されています。また、Visual C++ 6.0で

は、Win32アプリケーション（MFCを使わずにWindowsアプリケーションを作成する）やWin32コンソールアプリケーション（DOSプロンプトで動作するアプリケーションを作成する）などのプロジェクトタイプについてもAppWizardが追加されています。これによってプログラムのタイプに応じた最適なスケルトンが用意されるようになっています（ただし、本書ではMFC AppWizard以外のAppWizardについては触れない）。

そうはいっても、スケルトンはしょせんスケルトン。作成されたツールバーには、不必要な機能なら一杯あるのに、一番欲しい機能は抜けています。プリントアウトの機能が付けられても、肝心のデータを読み込む機能はなかったりします。

この先、骨格だけで足りない部分に肉付けをしていくのは、やはりプログラマの役目です。その作業を助けるために使用するのが、リソースエディタやClassWizardなどのツールなのです。

● リソースエディタ

Windowsプログラミングでは、しばしば**リソース**というものを利用します。リソースとは、簡単にいってしまえば、メニューやアイコン、ダイアログボックスなど、プログラムの中で利用されるデータパーツです。

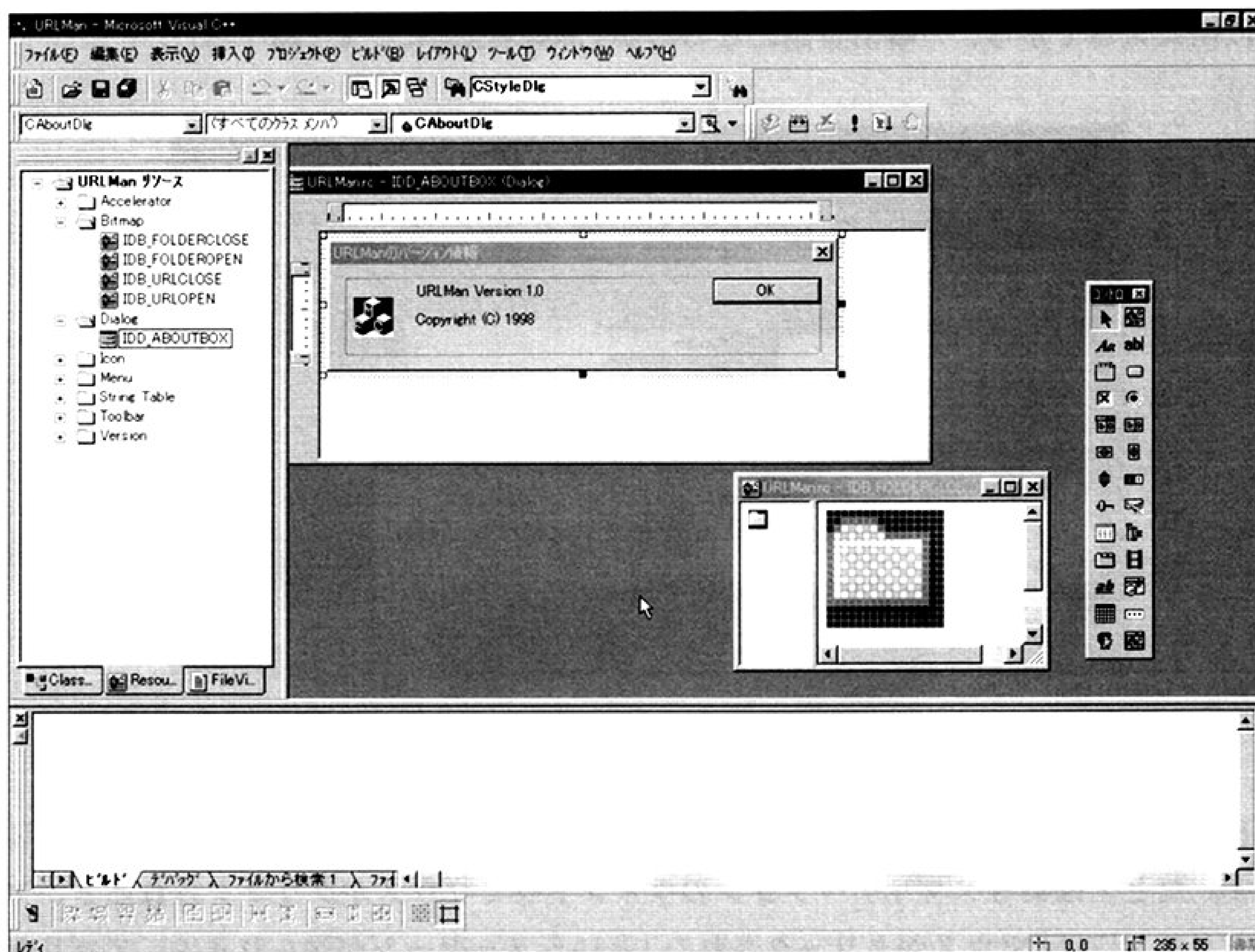


図1-4 リソースエディタ

リソース形式のデータは、プログラムとは別に作っておいて、あとから実行ファイルにバインド（Bind：結び付け）されます。データとプログラムコードを分離させるこの機構には、いくつかのメリットがあります。たとえば複雑なメニュー処理と複雑なプログラムがお互いの相乗効果で超複雑になるといった厄介ごとが自然に避けられますし、すでに完成しているプログラムのアイコンやメニューだけを差し替えるような芸当も可能です。

Developer Studioには、このリソースの作成や編集を行うリソースエディタが組み込まれています（**図1-4**）。実際のプログラミングにおいては、リソースエディタはスケルトンとして作成されたメニューに手を加えたり、新しいダイアログボックスを追加するのに利用します。さきほどもいいましたが、AppWizardが生成したスケルトンは、そのままでは何もできません。このスケルトンに肉付けをして、使いやすいインターフェイスを作成するためにリソースエディタはあるのです。

また、Developer Studioで、実行可能ファイル（.exeファイル、.dllファイルなど）を開いた場合には、リソースエディタを使用して、そのリソースを修正したり、取り出すことも可能です。

● ClassWizardとWizardBar

AppWizardとリソースエディタを使ったあとで、いよいよプログラムコードの記述という段階に入ってから、一番お世話になるのがClassWizardです（**図1-5**）。

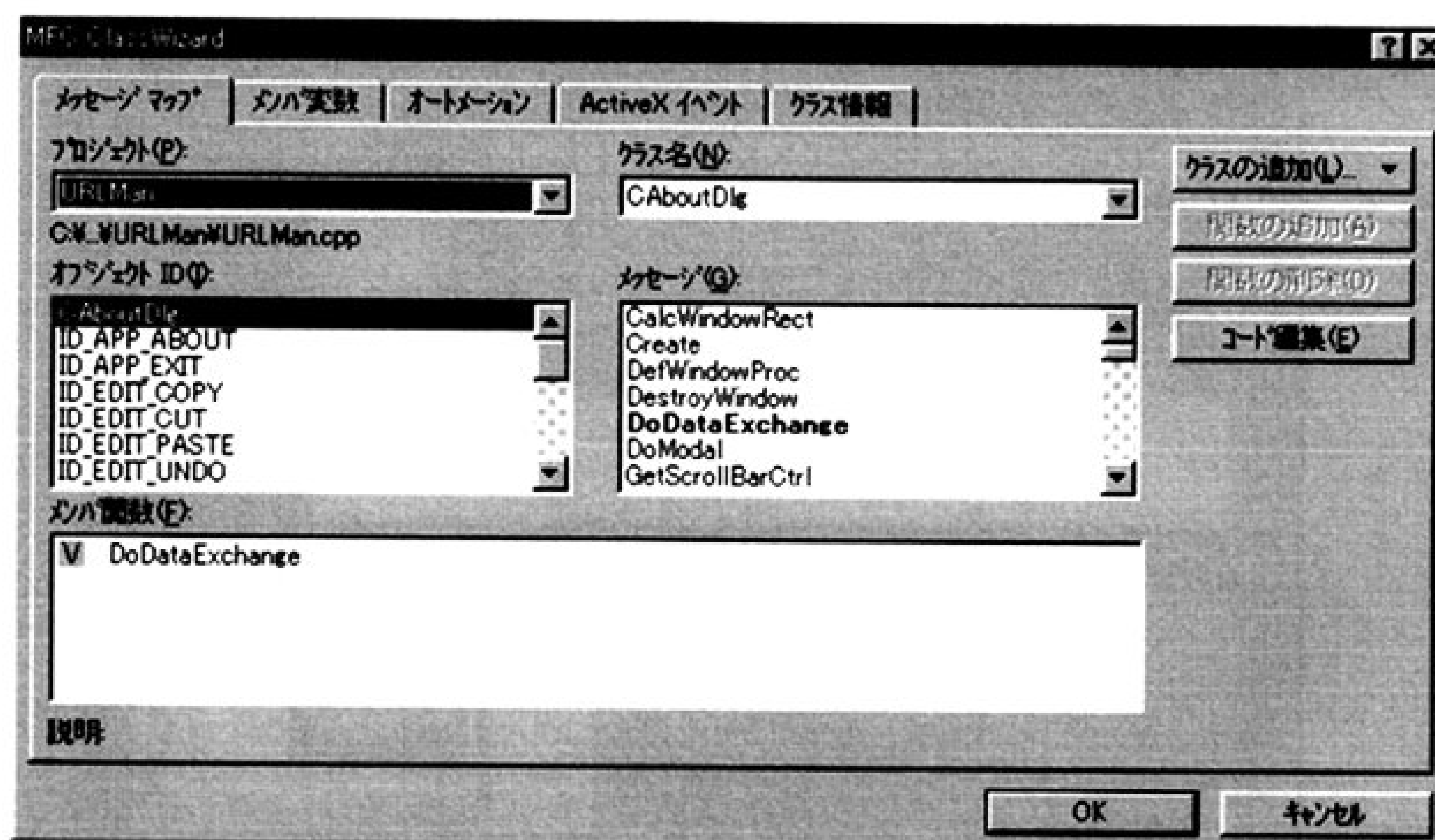


図1-5 ClassWizard

詳しくは3章で述べますが、Windowsアプリケーションの動作の基本は**メッセージ**にあります。たとえばあるアプリケーションのウィンドウでマウスがクリックされると、アプリケーションには「マウスがクリックされた」というメッセージが送られます。アプリケーションは、キーボードやメニュー、その他さまざまな場所からメッセージを受け取ります。それぞれのメッセージに対して、どのような対処を行うか（あるいは無視するか）を細かく

決めていけば、最終的にアプリケーションの動作が定まっていくわけです。

ClassWizard にはいくつかの役割がありますが、その中でもっとも大きなウェイトを占めるのが、まさにこの「メッセージごとの処理を決めていく」作業です。そのほかにも、クラスの新規作成、クラスへの変数の追加など、ClassWizard は後述する**クラス**というものにかかわるさまざまな操作も担当しています。クラスについてはやはり3章で説明することにしましょう。



図1-6 WizardBar

また、ClassWizard の機能を簡略化した **WizardBar** も用意されています(**図1-6**)。これを使用するとメッセージハンドラの作成/編集時にいちいち ClassWizard を起動する必要がなくなったり、.cpp ファイルの編集時にヘッダファイル(.h ファイル)をワンタッチで起動することができるようになり、プログラミングが一層快適に行えるようになります。また WizardBar はプログラマが編集中のソースコード位置を逐一監視していて、現在編集中のクラス名、関数名を常に最新の状態に保って表示します。おかげでプログラマは迷子にならずに済むうえ、<魔法の杖>ボタンを1つクリックするだけで注目しているクラスを操作できるというわけです。

● HTML ヘルプ

Visual C++には、すべてをハードディスクにインストールするのは勇気がいるほどのオンラインヘルプファイルが付属してきます。システムが巨大ならばドキュメントも膨大というわけです。この膨大なドキュメントのブラウザが HTML ヘルプです。

HTLM ヘルプを使った情報のブラウジングは非常に快適です。ウィンドウ左側に配置された目次をダブルクリックすれば指定したページが表示されますし、検索ダイアログボックスを使えば全文検索が可能です。また[戻る]、[次へ]、[ブックマーク]といった WWW ブラウザですでにおなじみの操作性は初めて使うユーザーにもわかりやすいものです。

HTML ヘルプは名前が示すように、ドキュメントのフォーマットには HTML が使われています(ただし、実際にはアーカイブファイルのように複数の HTML ファイルが1つの CHM ファイルにまとめられている)。HTML について改めて説明は必要ないでしょう。WWW で利用されているあの HTML フォーマットです。つまり HTML ヘルプは一種の WWW ブラウザといえます。もちろん http:で始まる URL を指定すれば、ネットワーク越しに最新のドキュメントを直接参照することも可能です。最近では Microsoft を始めとしたソフトウェアデベロッパが最新の情報を提供するために WWW を利用することはもはや当たり前です。もし Visual C++のドキュメントが Microsoft 独自のフォーマットで記

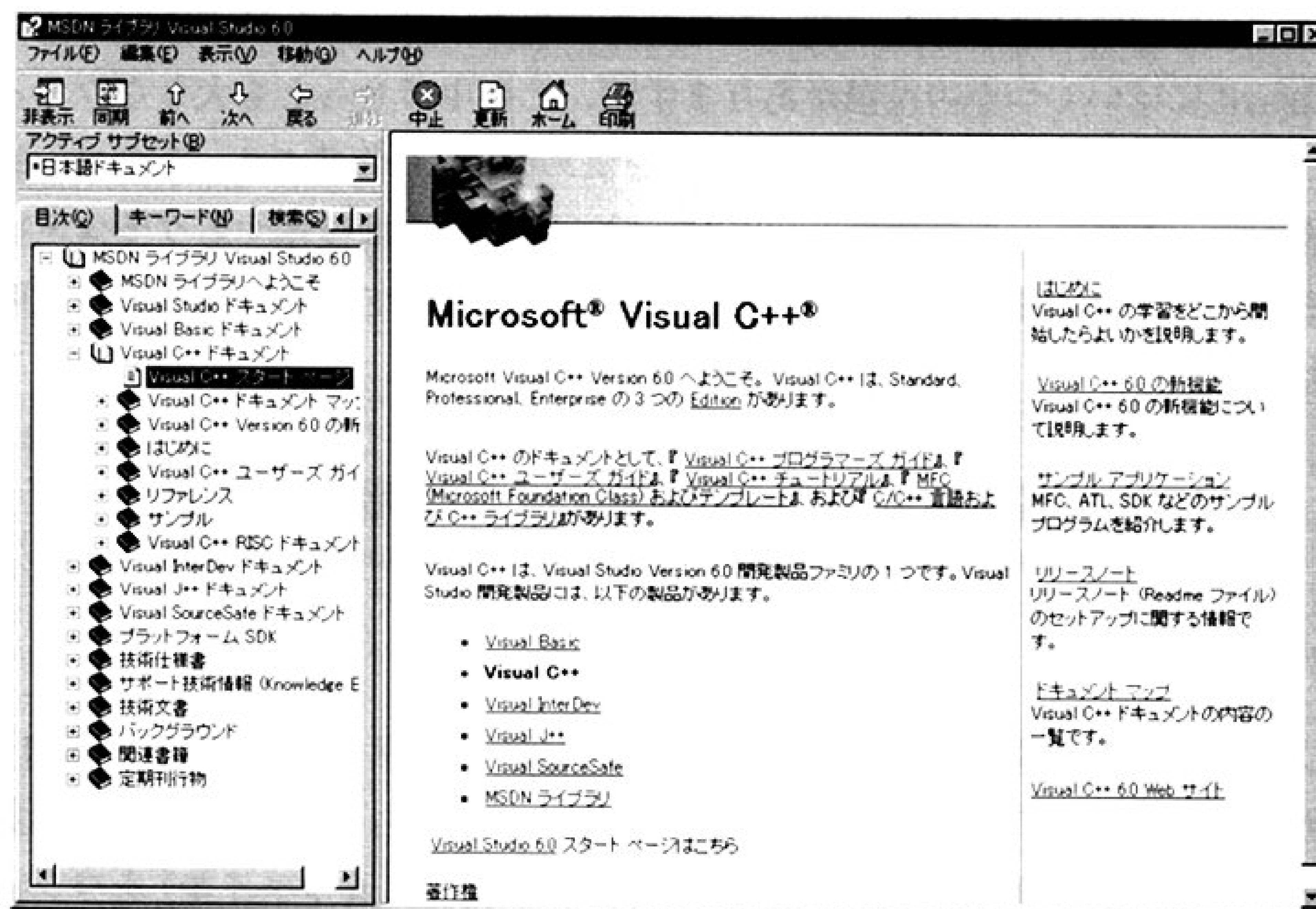


図1-7　HTMLヘルプ

述されていたならば、他社のドキュメントをシームレスに参照することは不可能だったでしょう。

またHTMLヘルプはWWWブラウザであるだけでなく、ActiveX Documentにも対応しています。ActiveX Documentが何物であるかはともかく、おかげでHTMLヘルプにはHTMLドキュメントだけでなく、Microsoft WordやExcelといったActiveX Documentに対応しているアプリケーションのドキュメントをも表示できるようになったのです。つまりプログラマがWordを使って書いた仕様書をHTMLヘルプに表示することもできるということです。

● デバッガとブラウザ

ここまで紹介してきたツールは、プログラム開発のサポート役であり、Visual C++の「光」の面を代表する存在でした。しかし皆さんご存じのとおり、プログラミングという作業には、避けては通ることのできない「闇」もあります。つまりプログラムのデバッグです。

Visual C++では、AppWizard、リソースエディタ、ClassWizardなどのツールがプログラミングを助けてくれる代わりにプログラマの目の届かない部分も多く、その動作を理解するのはたいへん面倒な作業となります。プログラマの関知しない部分で作成されたコードが実際にはどう動いているのかを隠してしまうのです。

そこでVisual C++には、プログラムの動作の解析を助ける2つの機能、非常に強力なデバッガとブラウザが用意されました(**図1-8**)。

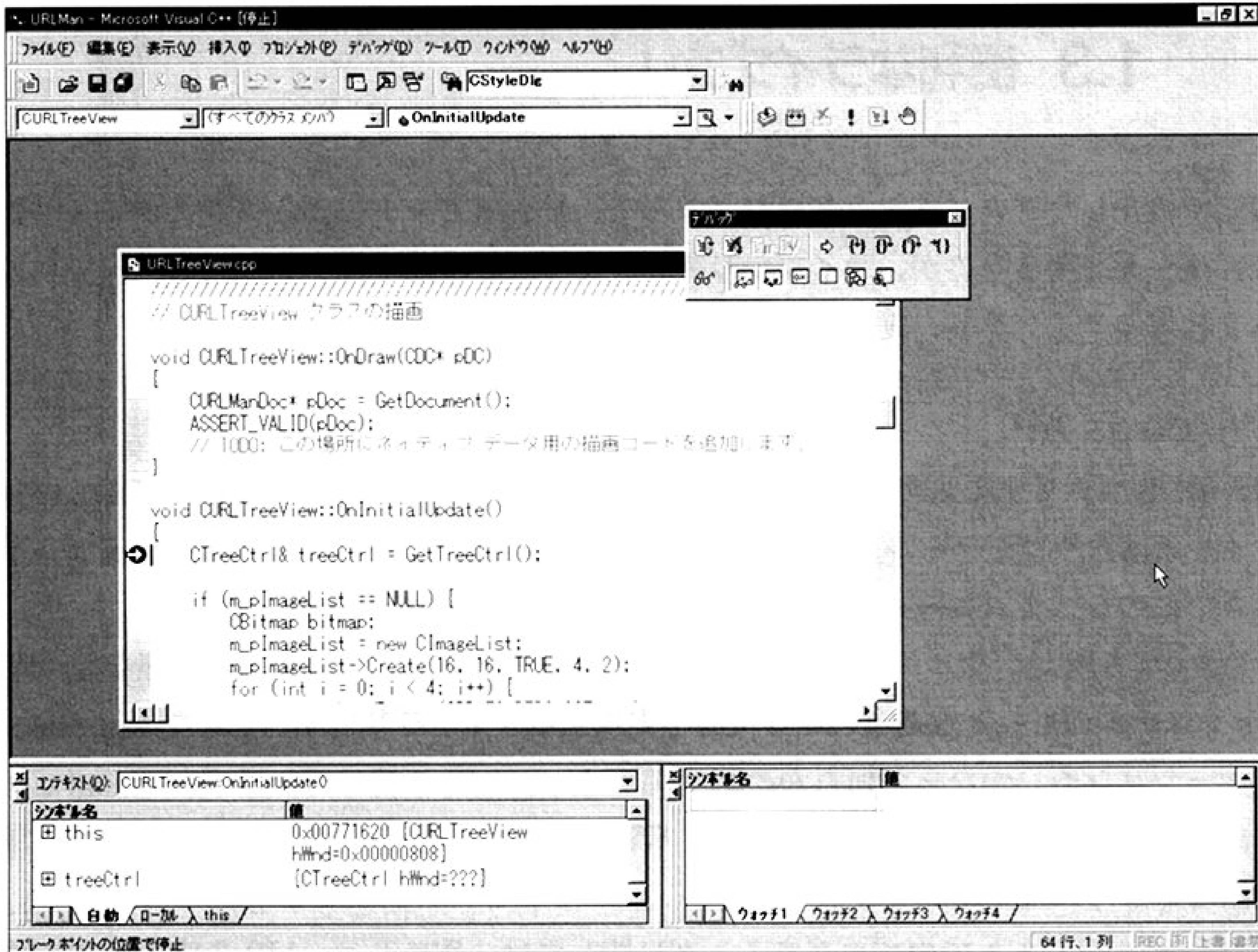


図 1-8 デバッガを使用しているところ

デバッガを使用するとプログラムを1行ずつ動作させたり、指定した位置まで一気に実行したりすることができます。特定の変数の値を表示しながらプログラムを実行することもできます。一方ブラウザは、関数の呼び出し階層や、変数や定数の定義位置と参照位置、さらに後述するクラス階層など、静的なプログラムの構造を見せてくれます。

デバッガもブラウザも、現在作成中のプログラムだけでなく、Visual C++の標準インクルードファイルまでさかのぼってデータを示してくれるところに特徴があります。このおかげで、プログラマはAppWizardが「勝手」に作り出したプログラムの流れや静的構造をとことん追究することができます。

本書で扱うプログラムでは、デバッガやブラウザの出番はほとんどありませんが、Visual C++に付属してくるサンプルプログラムなどを解析する場合には、このデバッガとブラウザが強い味方となってくれるでしょう。

1.3 便利なライブラリ

前節で紹介したプログラミング環境と並んで、Visual C++によるプログラミングの大きなメリットとなるものが、**ライブラリ**の存在です。この節では、Visual C++で利用できる2種類のライブラリ、Windows APIとMFCの役割を簡単に説明します。

Windows API

MS-DOSプログラミングでは、システムコール（INT 21h）を使って、MS-DOSの持つファイル入出力やコンソール入出力機能を呼び出すことができました。Windowsプログラミングでこのシステムコールと同様な役割を果たすものが**Windows API**です。

Windows APIは、ウィンドウの作成や表示、データ出力、マウス入力の処理など、Windowsシステムのもっとも基本となる機能を実現するライブラリです。どんなWindowsアプリケーションも、かならず何らかの形でWindows APIを使っています。Visual C++で作成するアプリケーションの中からも、Windows APIはライブラリ関数形式で簡単に呼び出せるようになっています。

ただしVisual C++でのプログラミングに関しては、Windows APIを直接呼び出すことはあまりありません。これはMS-DOSプログラミングにおいて、たとえばファイルをオープンする際に、システムコールのopen関数ではなく標準ライブラリのfopen関数が使われることと似ています。つまり、Windows APIというものは、システム寄りの低レベルな部分で動作するため、Windowsシステムの動作を熟知していなければ使いこなせないのです。

クラスライブラリMFC

そこでVisual C++では、プログラマの負担を減らし、Windowsシステムの機能がもっと簡単に利用できるように、Windows APIの上に一枚かぶさって動作するライブラリが用意されました。これが**Microsoft Foundation Class（MFC）**です。MFCを有効に活用するためにVisual C++は生まれたのだといっても過言ではないほどMFCとVisual C++は密接に結び付いています。

Windows APIには、非常に多くの関数が用意されていますが、それぞれが独立した関数であるため、どの関数がどんな処理を行うものかがわかりにくくなっていました。これに対して、MFCではクラスというものを使って関数を処理内容によって分類しているので、必要な機能を実現するための関数が見つけやすくなっています。

Visual C++が提供する各種ツールとMFCを利用することによってWindowsプログラミングはWindows APIをそのまま利用していた頃よりも格段に簡単になるはずです。

1.4 What's new?

ここまでVisual C＋＋ 6.0の概要を紹介してきましたが、従来からのVisual C＋＋ユーザーから「もうわかったからどこが変わったんだ？」という不満の声が聞こえてきそうなので、ここでまとめておきましょう。

Visual C＋＋ 6.0を一言で表現すれば「熟成バージョン」と呼べるでしょう。劇的に何かが変わったわけではなく、広範囲に渡って機能追加や機能拡張が行われています。今回Visual C＋＋がバージョンアップされた1つの契機として Windows 98のリリースがあげられますが、Visual C++ 6.0が大きくWindows 98に依存しているわけではありません。付属するサンプルプログラムを見ても、Windows 98でなければ動作しないプログラムはごく少数にすぎません。Windows 98で標準装備された Internet Explorer 4.0（以下、IE 4.0）を利用したプログラミング環境が強力にサポートされている点は Visual C++ 6.0の大きな魅力ですが、これもWindows 95にIE 4.0をインストールすれば済むだけの話です。したがって、Visual C＋＋ 6.0での変更点を解説するには細かなトピックを並べることになってしまいます。しかし、これはオンラインマニュアルに詳細に解説されているので、ここでは主だった変更点に絞って簡単に紹介するにとどめておきます。

まずDeveloper Studioですが、見た目はほとんど変わっていません。しかし熟成バージョンだけあって（勝手にそう呼んでいるだけですが）、地味ながら非常に実践的な機能が2つ追加されています。

1つは**IntelliSense**と呼ばれる、テキストエディタでの補完入力機能です。これはWindowsプログラミングにありがちな、なが〜い識別子であっても、途中まで入力すれば残りは自動的に補って入力してくれるありがたい機能です。たとえば、「CCachedDataPathProperty」と入力したいとしましょう（これはMFCで定義されているクラスの名前です）。この場合、「CCached」まで入力して[Ctrl]＋[ ]を入力すれば、残りの「DataPathProperty」は自動的に入力されてしまいます。もし早めに[Ctrl]＋[ ]を押してしまって、ほかにも補完候補があるといった場合には、リストボックスがポップアップされるので、リストから望みのものを選択するだけで済みます。さらに、オブジェクト名に続けてピリオドを入力したときには、そのオブジェクトのメンバ一覧が表示されるなど、さまざまな場面で補完機能はプログラマを助けてくれます。入力作業が省略できる上に、タイプミスも減らせる、実にありがたい機能です。

もう1つは、ClassViewの動的解析機能です。ワークスペースウィンドウに表示されているClassViewには、プロジェクトに含まれているクラスとそのメンバがツリー状に表示されます。これまでClassViewの更新は編集中のファイルを保存したときに行われていましたが、Visual C＋＋ 6.0からはファイルへの保存を待つことなく、入力した直後にその

結果がClassViewに反映されるようになりました。したがって、ClassViewには常に現在の状態が維持されるようになったわけです。

また、周辺ツールの拡充もVisual C++ 6.0の目玉の1つです。まずもっとも頻繁に利用するMFC AppWizardですが、ドキュメント－ビュー・アーキテクチャを使わないスケルトンを生成することができるようになりました。従来、MFCを使ったWindowsアプリケーションを作成するには、AppWizardを利用する限り、ドキュメント－ビュー・アーキテクチャに従わざるをえませんでした。ドキュメント－ビュー・アーキテクチャはある程度以上の規模で、複雑なドキュメント（アプリケーションが扱うデータ）を扱うアプリケーションならば効果はあるものの、ちょっとしたコンパクトなアプリケーションを作るには、必要以上に複雑でいささか面倒なアーキテクチャです。これまでMFCは使いたいけれど、ドキュメント－ビュー・アーキテクチャは使いたくない、というジレンマを感じたプログラマは多いと思いますが、Visual C++ 6.0でようやくこれが解消されました。ドキュメント－ビュー・アーキテクチャを使用しないアプリケーションについては、Appendix Cで説明をします。

MFC AppWizardのもう1つの拡張機能は、エクスプローラスタイルのアプリケーション用スケルトンを作成する機能です。エクスプローラスタイルとは、ウィンドウを左右に分割し、データを階層的に分類して左側にツリー表示し、ここで選択したデータを右側のウィンドウに表示するという、まさにエクスプローラで使われているスタイルのことです。エクスプローラのようにファイルアクセスに使うだけでなく、最近ではメーラや住所録でも使われていることからもわかるように、このスタイルのアプリケーションは応用範囲の広いものです。今まででもエクスプローラスタイルのアプリケーションを作ることはできましたが、AppWizardのサポートによって、より簡単に作成できるようになりました。これについては、第4部で触れます。

次にライブラリに目を移しましょう。Visual C++のバージョンアップに伴って拡張を繰り返してきたMFCですが、最近では大幅な変更は行われず、Windowsシステムの新機能を取り入れる程度に落ち着いてきました。今回のバージョンアップで、MFCこそ6.0へとバージョンアップしましたが、使われるDLLのファイル名はMFC42.DLLです。メンバ関数の拡張やバグフィックスは広範囲に行われているものの、バージョン番号が示すとおり、新規クラスの追加は、ほとんどがIE 4.0に関連するものだけです。簡単に紹介しておくと、CHtmlViewクラス（HTMLファイルを表示可能なビュークラス。インターネットへのアクセス機能も持つ）、CReBarクラス（IE 4.0で導入された移動可能ツールバー）、CMonthCalCtrlクラス（カレンダーを表示する、日付の入力に使うコントロールを扱うクラス）などが追加されたクラスです。

最後にオンラインドキュメントについて触れておきましょう。これは見た目としてはもっとも大きな変更点でしょう。Visual C++ 5.0までは、Developer Studioに組み込まれたInfoViewerがドキュメントブラウザとして機能していましたが、Visual C++ 6.0からは

Windows 98 以降の Windows 標準ヘルプシステムである HTML ヘルプがこれにとって変わることになりました。HTML ヘルプは Developer Studio から分離独立して単体で機能するアプリケーションですが、Developer Studio での編集中に F1 キーなどで呼び出せるように、連携して動作するところは変わりません。またドキュメントの目次がツリー状に表示されたり、キーワード検索／全文検索が可能なところも変わっていません。すでに Visual C＋＋ 5.0 から HTML をベースとしたヘルプシステムに移行していたことを考えると、単純に従来の InfoViewer がウィンドウを独立したものと考えてよいでしょう。近年 PC の性能が向上して画面が広くなり、また Windows 98 では複数のディスプレイモニタを併用するマルチモニタがサポートされたことを考えると、広大な画面を有効に使って、Developer Studio を開きながら同時にヘルプを参照できるようになったことはプログラマにとって喜ばしい（？）改善といえるでしょう。

1.5 最後にひとこと

ここまで「Visual C＋＋で未来はバラ色！」的なことばかりを取り上げてきましたが、本当のところはこんな簡単にいくはずがないということだけは気に留めておいてください。実際にプログラムを組む段になってから、「ハテ？」と首を傾げてしまうことが何度もあるかもしれません。

Visual C＋＋に限らず、大規模なクラスライブラリを利用した開発環境は、ほとんどの場合、最初の取っ掛かりが非常に悪いのです。そして、クラスライブラリを使いこなすには、その設計思想や影に隠れた部分のプログラムの実際の動作などをしっかり理解することが必要になります。理解ができるまでは、霧の中をさまよっているような気分を味わうことになるかもしれません。そしていつ晴れるかもわからない霧の中を進む作業は、面倒で辛いものになるでしょう。しかし、ライブラリの使いこなし方がわかってくれば、プログラミングはかならず効率的に、簡単に、そして楽しく行えるようになるはずです。

それでは、Visual C＋＋と MFC を使いこなすべく、2 章以降では実際にプログラムを作成しながら基本となる知識を身に付けていくことにしましょう。

2 Visual C++流プログラミング!!

1章ではVisual C++とその世界をひととおり探検してもらいました。この章ではいよいよVisual C++を利用したプログラミングの世界に足を踏み入れることにします。

その手始めに、まずはVisual C++が提供する優れたツールの1つ、MFC AppWizardを使ったプログラムの自動生成を体験してみましょう。Visual C++によって、Windowsアプリケーションの作成がいかに単純化されるのか、実感してください。

そして次に、プログラミング言語入門といえばお約束の、"Hello World"プログラムを作成します。ここでは普通の画面表示のほかに、メッセージボックスを利用した表示法にも挑戦します。MFC AppWizard／リソースエディタ／ClassWizardを連携させた、Visual C++の典型的なプログラミング手順を見てください。

Visual C++ 6.0ではMFC AppWizard以外にも多くのAppWizardが提供されていますが、本書ではもっとも基本的なAppWizardであるMFC AppWizardだけを利用するので、以降ではMFC AppWizardのことを単にAppWizardと呼ぶことにします。

2.1 プロジェクトジェネレータAppWizard

Visual C++で作成するアプリケーションは、非常に多くのファイルから構成されます。たとえば、プログラムコードが書かれたソースファイルや、コンパイルのための情報が書かれたメイクファイル、全体の設定を記述したプロジェクトワークスペースファイル、さらにアプリケーションのタイトルバーに現れるアイコンのデータファイルなども含まれます。

Visual C++ではこれらをひとまとめにして**プロジェクト**と呼びます。アプリケーションの開発は、基本的にはこのプロジェクトの中ですべて行われます。簡単にいってしまえば、「プロジェクト名＝アプリケーション名」と思って間違いないでしょう。

AppWizardはプロジェクトの基本構成を設定するツールです。設定内容には、たとえば以下のようなものがあります。アプリケーションの全体像に合わせて、これらの条件を

設定すると、AppWizardはそれに対応したアプリケーションの枠組みを設計し、必要なすべてのファイルをプロジェクトに組み込んでくれます。

- **プロジェクトワークスペース名**（すなわちアプリケーションの名前）
- **プロジェクトを格納するフォルダ**（ディレクトリ）
- **アプリケーションの形式**
- **ウィンドウの細部の構成**（ツールバーやステータスバーの有無）
- **その他**

AppWizardでプロジェクトの枠組みを決める

それでは実際にAppWizardを利用して新しいプロジェクトを作成してみましょう。

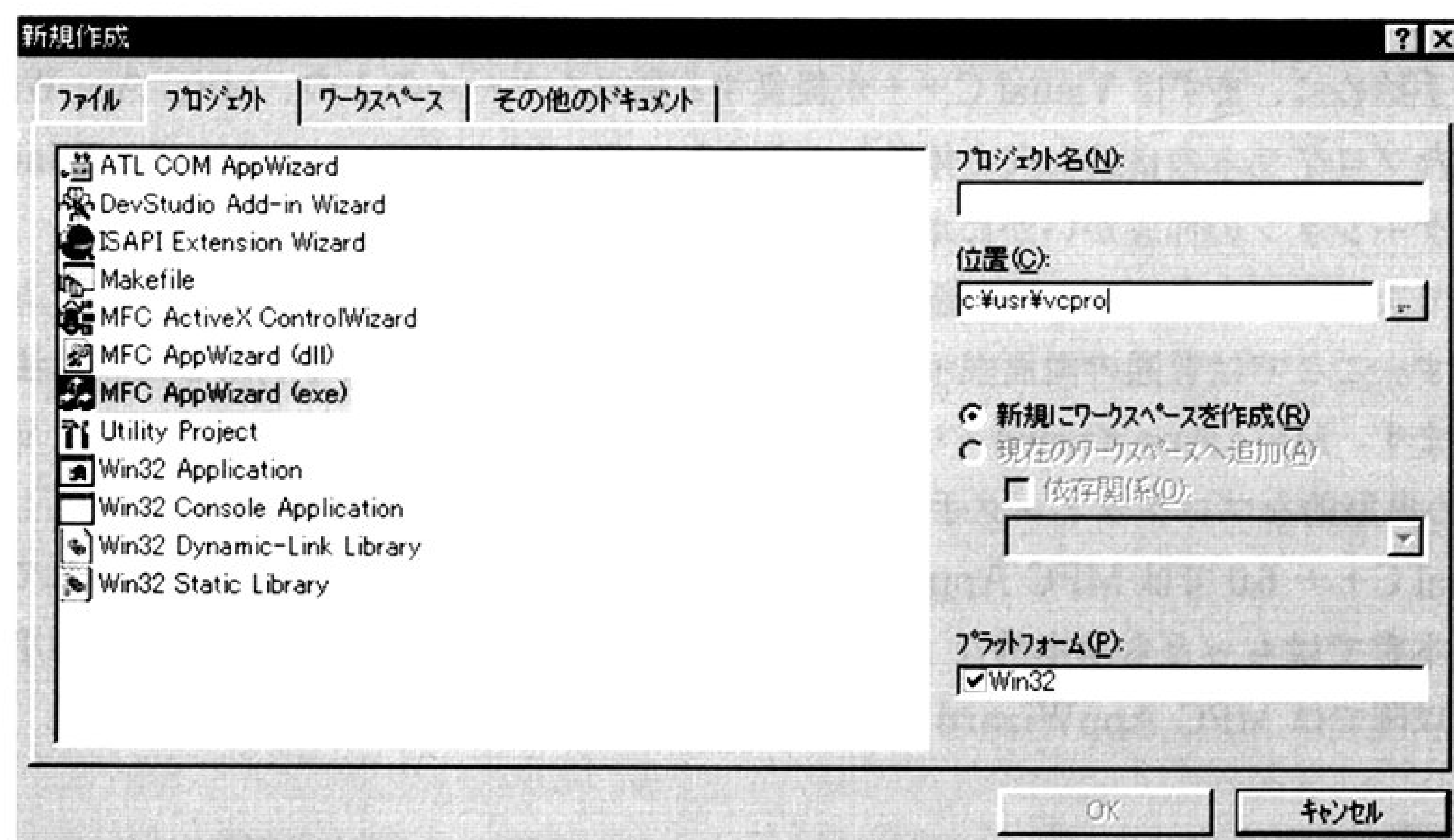


図2-1 プロジェクトワークスペースの新規作成

まずDeveloper Studioのメニューから［ファイル］－［新規作成］を選んでください。すると［新規作成］ダイアログボックスが表示されるので（**図2-1**）、ここで［プロジェクト］タブの［MFC AppWizard (exe)］を選択してください。ここにはプロジェクト作成をサポートするいくつものツールが用意されていますが、［MFC AppWizard (exe)］がもっとも基本的、かつ一般的な手段です。［MFC AppWizard (exe)］を使うと、MFCを使ってWindowsアプリケーションを作成するために欠かせない、さまざまなセットアップが行われます。このセットアップはソースファイルのスケルトンやリソースファイル、ワークスペースファイル（Developer Studioが使う作業環境の設定ファイル）など広範囲に及ぶため、AppWizard以外の手段でMFCアプリケーションを作成するにはかなりの困難が伴います。Visual C++を深く理解するまではAppWizardのサポートは欠かせません。

［MFC AppWizard（exe）］を選択したあとには、同じく［新規作成］ダイアログボックスで［プロジェクト名］と［位置］、それに［プラットフォーム］を指定します。

［プロジェクト名］は最終的に作成されるアプリケーションの名前であり、AppWizardが自動生成するファイル名やソースコード中の各種の名前などのベースともなります。今後のプログラミング作業に密接にかかわるものですから、よく考えて指定してください。今回は［Test］というプロジェクト名にしておきましょう（**図 2-2**）。

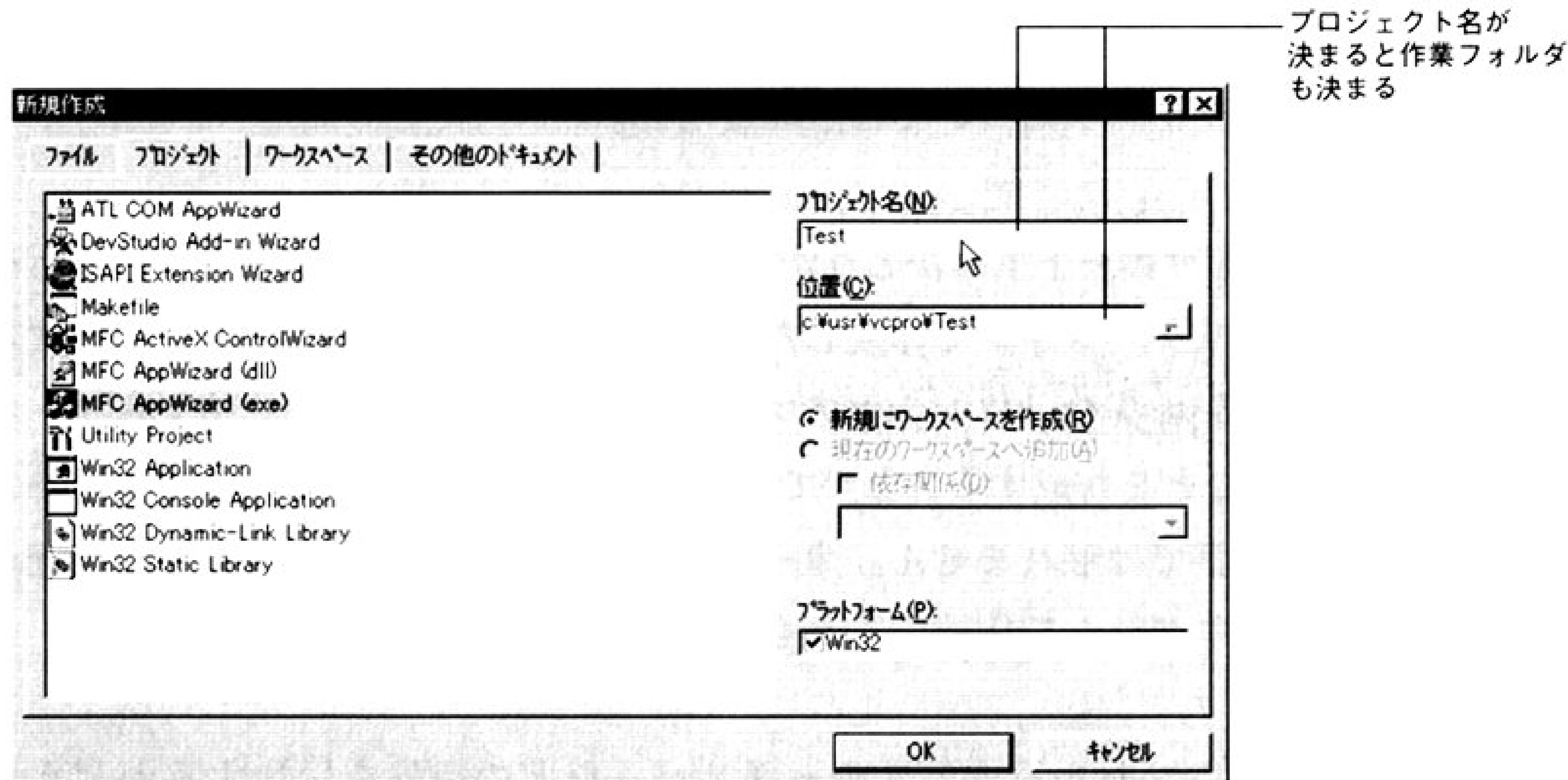


図 2-2　プロジェクト名、フォルダを指定したところ

プロジェクトを格納する［位置］を指定するには、隣にある<...>ボタンをクリックして、［ディレクトリの選択］ダイアログボックスから適当なフォルダ（ディレクトリ）をマウスで選択します。一般的には、あらかじめ専用のフォルダを用意しておくとよいでしょう。［位置］に指定したフォルダの下に、プロジェクトごとにフォルダが作られ、そこにプロジェクトを構成するすべてのファイルが格納されます。たとえばここで「C:¥usr¥vcpro」というフォルダを選んだ場合は、プロジェクト名が「Test」なので、「C:¥usr¥vcpro¥Test」というプロジェクト用のフォルダが作成されます。なお、以降本書では作業用フォルダとして「C:¥usr¥vcpro」を利用することにします。

最後にプラットフォームですが、ほとんどの読者の環境では［Win32］だけが表示され、さらにこれにチェックが付いているでしょうから、なにもいじることはありません。ここにその他の選択肢が現れるのは、Windows CE 用開発キットなど、Windows95/NT 以外の OS で動作するアプリケーションを開発するためのアドオンキットをインストールしている場合です。通常は気にする必要はありません。

プロジェクトのタイプと名前、それに位置（フォルダ）の設定が終了したら、<OK>ボタンをクリックしてください。するとここで初めて AppWizard が起動され、AppWizard による問診が始まります。AppWizard で設定することができるのは、次のような要素です。

●ステップ 1

作成するアプリケーションのタイプとリソースで使用する言語の種類を指定します。さらに、ダイアログベースのアプリケーションも作成することができます。リソースに使用する言語は日本語か英語を指定することができます。また、SDI/MDI形式を選択した場合には、ここでドキュメントービュー・アーキテクチャをサポートするかどうかを指定します。

●ステップ 2

データベースのサポートを行うかどうかを指定します。なお、本書ではVisual C++でのデータベースサポートについては触れません。

●ステップ 3

複合ドキュメントをサポートするかどうかを指定します。複合ドキュメントをサポートすると、他のアプリケーションで作成したドキュメントやコントロールを格納するコンテナアプリケーションや、コンテナアプリケーションにデータを供給するサーバーアプリケーションなどを作成することができるようになります。ただし、複合ドキュメントについては本書では触れません。オートメーションをサポートするか、ActiveXコントロールをプログラムで使用するかもここで決定します。

●ステップ 4

アプリケーションにツールバー、ステータスバーなどの機能を付加するかどうかを指定します。ツールバーに関しては、従来のスタイルのものを使用するか、Internet Explorerで使用されているReBarを使用するかも指定できます。MAPI/Windowsソケットをサポートするかについてもここで指定します。さらに<詳細設定>ボタンをクリックした場合には、ウィンドウのスタイルやアプリケーションに対応付ける拡張子の種類などの指定を行うことも可能です。

●ステップ 5

SDI/MDI形式のアプリケーションを作成する場合、その外観を標準的なウィンドウ形式にするか、エクスプローラ的名ものにするかをここで指定します。AppWizardやClassWizardが生成するコードにコメントを付加するかを指定します。また、Professional Edition以上のバージョンでは、ここでリンクするMFCライブラリの種類(共有DLL/スタティックライブラリ)を選択できます。

●ステップ 6

AppWizardが生成するクラスの名称と、それを保存するファイルのファイル名などを指定します。

以上の問診に対して回答していくことによってどのようなアプリケーションを作るのかを決めていくのですが、今回はすべて標準設定のままでかまわないので難しいことはありません(つまり、AppWizardが示した設定に対して、すべて「はい」と応えるわけです)。

選択肢はすべてそのままで＜次へ＞ボタンを5回クリックしてください。すると最後に、＜次へ＞ボタンがクリックできなくなって、問診が終わったことがわかります。

もし今までの作業で設定ミスなどがあった場合、＜戻る＞ボタンをクリックすれば、いつでも前の画面に戻ることができます。今回は何も変更することがありませんから、このままで結構です。では＜終了＞ボタンを選択し、AppWizard に仕事をしてもらうことにしましょう。すると、これから作成するアプリケーションに関する情報が**図 2-3** のように表示されます。

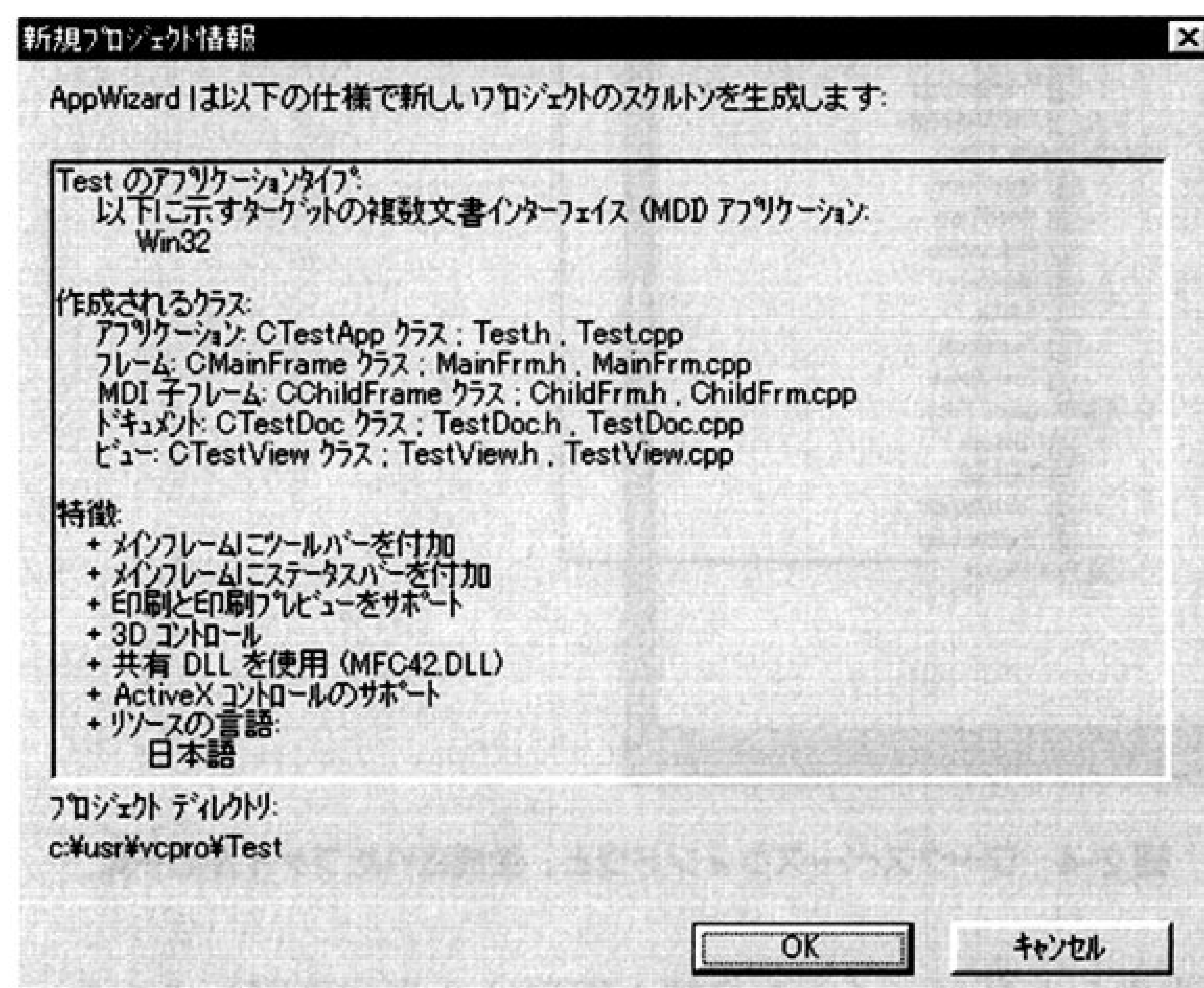


図 2-3 作成するアプリケーションの情報

ここが AppWizard の最後の質問です。ここでも＜キャンセル＞ボタンをクリックすれば、前の画面に戻ることができます。よければ＜OK＞ボタンをクリックして、実際にプロジェクトを生成してもらいましょう。

プロジェクトの構築が進むにつれて、ファイル名が次々と表示され、それが終了すると Developer Studio に戻ります。これだけの動作で Windows のアプリケーションに必要なソースのほとんどが揃ってしまいました。どんなファイルが作られたのか、ちょっと見てみましょう。メニューから［表示］－［ワークスペース］を実行すると、ワークスペースウィンドウが表示されます。さきほども説明したように、このウィンドウではウィンドウ下部にぶらさがっているタブを操作することによって、プロジェクトに含まれるクラス、リソース、ファイルを切り替えながら表示することができます。

AppWizard が生成したファイルを表示するには、［FileView］タブをクリックしてください。すると［Test ファイル］というフォルダが1つ表示されているので、これをダブルクリックしてください。続けて新たに表示された［Source Files］、［Header Files］、

[Resource Files] フォルダも同じようにダブルクリックして開くと、プロジェクトに含まれるすべてのファイルが表示されます。AppWizard のおかげで、これらのファイルが自動的に生成されたのです (**図 2-4**)。AppWizard とはアプリケーション (= App) の魔法使い (= Wizard) のことなのです。

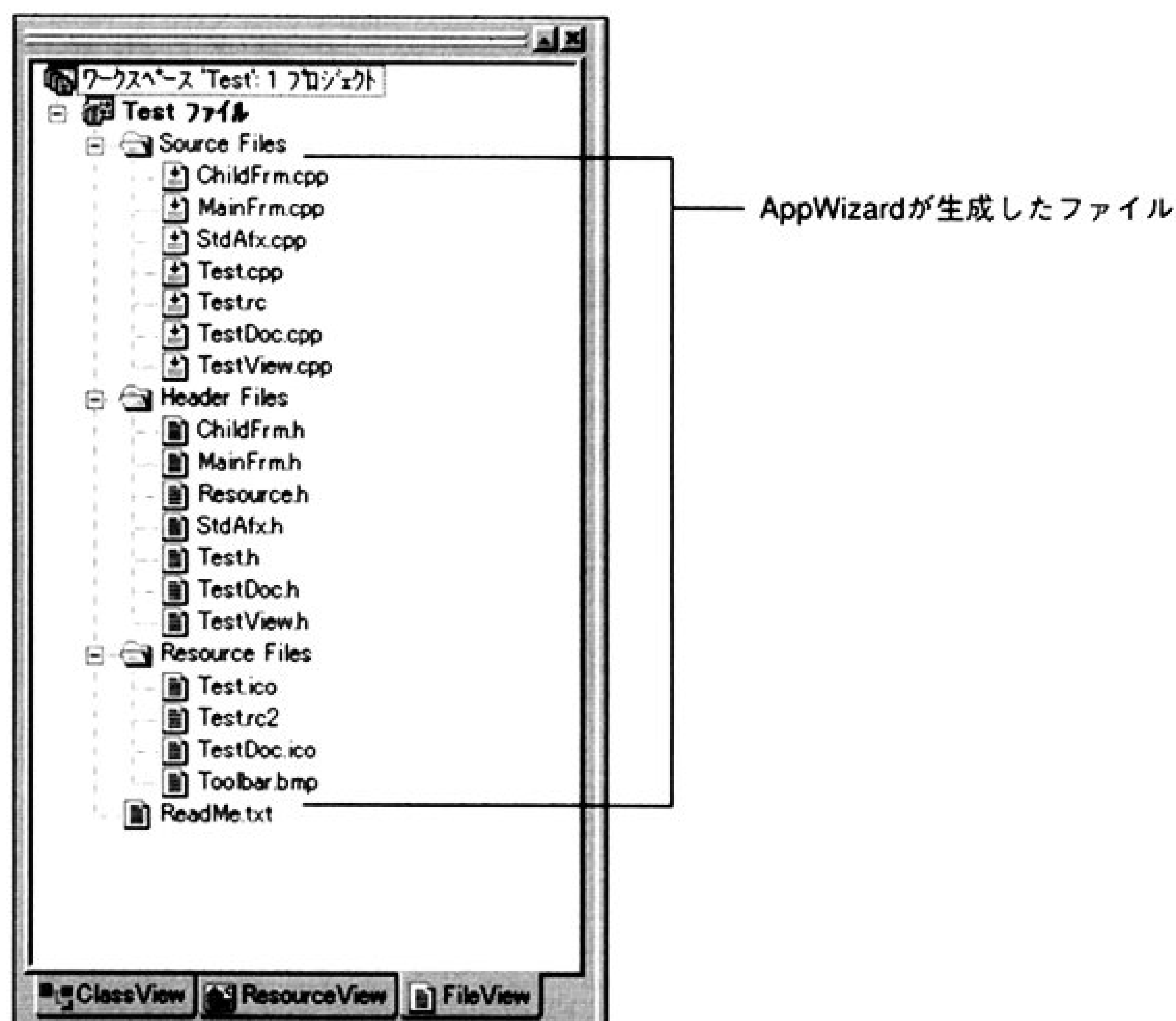


図 2-4 ワークスペースウィンドウと、生成されたファイルの一覧

AppWizard が生成した各ファイルの意味や用途は 3 章で説明しますが、プロジェクトと一緒に作られた ReadMe.txt にも概要が書かれていますから、一度このファイルに目を通しておくとよいでしょう。ワークスペースウィンドウに表示されている [ReadMe.txt] をダブルクリックすると ReadMe.txt の内容を読むことができます。今の段階では、ReadMe.txt を見ても理解できないことが多いかもしれません。しかしこの先ページを重ねていくうちに、だんだんプロジェクト内でのファイルの役割分担が見えてくるはずです。

AppWizard が出力したファイルの中には、アプリケーションの雛型となるソースコード一式も含まれています。すでに述べたように、プログラムの骨組みという意味から、これを**スケルトン**と呼びます。

スケルトンは、これから作るアプリケーションの出発点となるものですが、Windows アプリケーションとして最低限の機能を備えた一人前のプログラムでもあります。次の項ではこのスケルトンをコンパイルして、Visual C++ が提供してくれるアプリケーションの枠組みを簡単に眺めてみることにします。

●ボタン一発コンパイル！

ツールバーの<ビルド>ボタンをクリックすると、プログラムのコンパイルが始まります（メニューをお好みの方は［ビルド］－［ビルド］でも可）。このとき Developer Studio の画面には［アウトプット］ウィンドウが開き、コンパイル中のメッセージが表示されます。もしコンパイル中にエラーや警告があるとここに表示されます（**図 2-5**）。

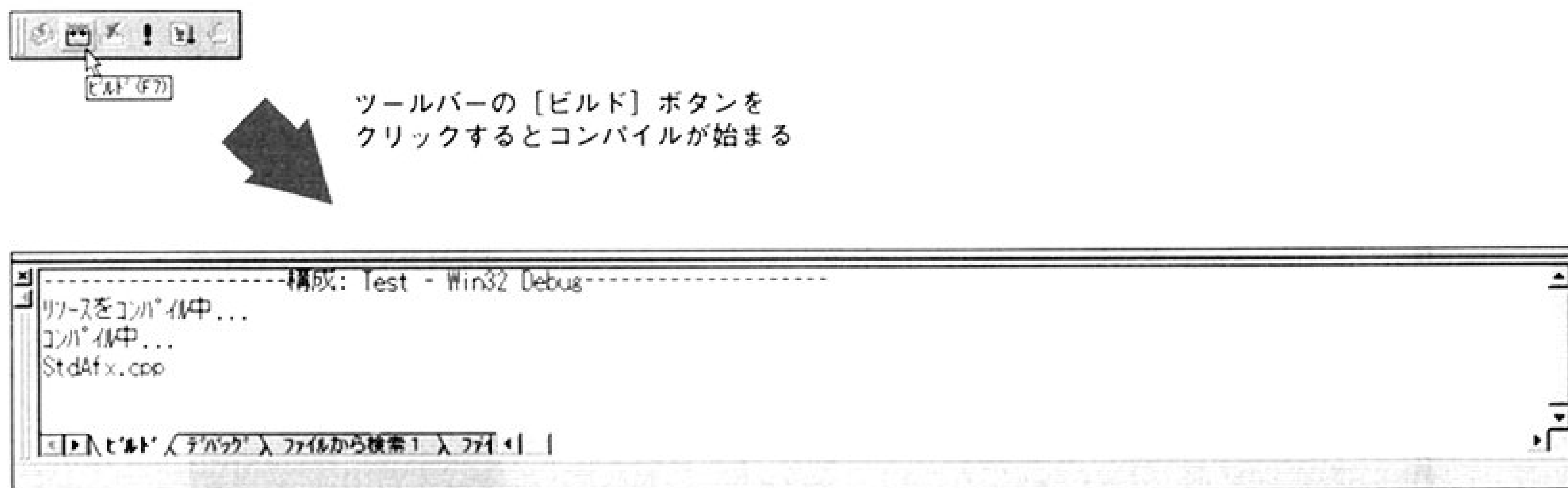


図 2-5 <ビルド>ボタンと［アウトプット］ウィンドウ

どうですか、**図 2-6** のように表示されましたか？ 今回は AppWizard が生成したスケルトンを変更を加えずにコンパイルしているので、何事もなく終了するはずです。最後に「Test.exe － エラー 0、警告 0」と表示されたらコンパイル終了です。

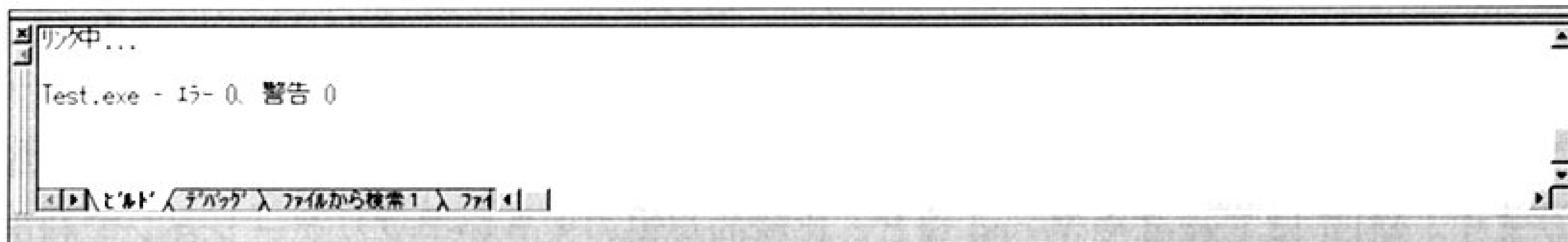


図 2-6 コンパイル終了の画面

コンパイルの途中では、スケルトンプログラムを構成するいくつかのファイル名が順に表示されます。いままでのプログラミング環境では、このようなモジュールに分割されたプログラムのコンパイルにはメイクファイルの作成や保守という面倒な作業が不可欠でした。しかし Visual C＋＋を利用すれば、そんな苦労も過去のもの。Visual C＋＋はプロジェクトに含まれる複数のファイルの依存関係を常に把握し、その最適なコンパイル方法を知っていますから（メイクファイルは Visual C＋＋が自分で作成する）、プロジェクトがいくつのファイルで構成されていても、<ビルド>ボタン一発で、簡単にコンパイルが実行できるのです。

それではコンパイルされたプログラムを起動してみましょう。ツールバーの<実行>ボタン（書類の隣に矢印があるボタン）をクリックしてください。プログラムが立ち上がるまでには、多少の時間がかかるかもしれません。これは、あとで説明しますが、今回のプロ

ジェクトはデバッグモードでコンパイルしたため、アプリケーションの実行ファイルに膨大なデバッグ情報が含まれているからです*1。

プログラムが起動すると、**図2-7**のように「Test - Test1」とタイトルのついたウィンドウが開きます。そしてその中に「Test1」というウィンドウが開かれましたね。

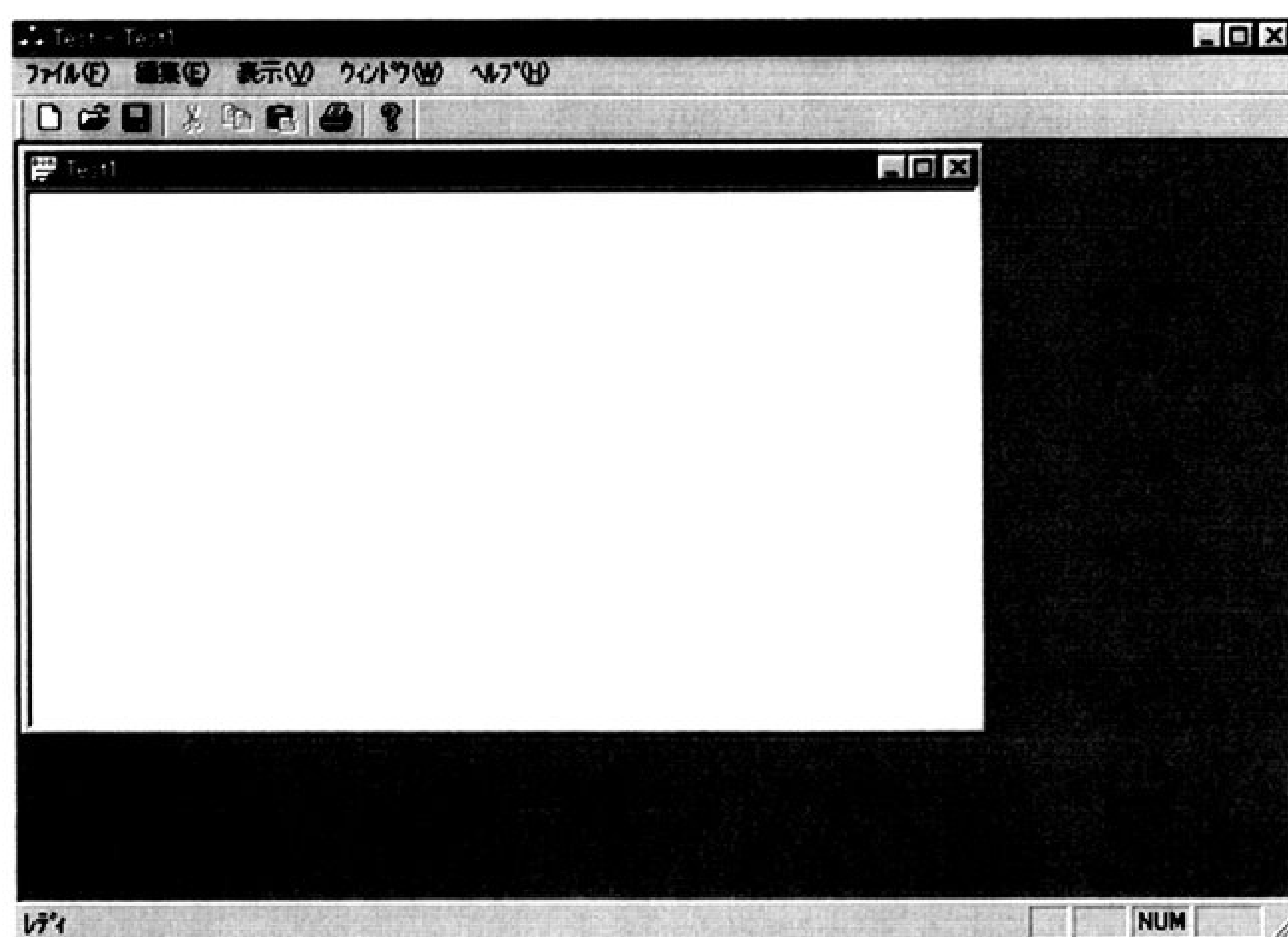


図2-7 プログラムの実行画面

このプログラムのように、1つのウィンドウ(親ウィンドウ)の中に別のウィンドウ(子ウィンドウ)を表示するアプリケーションのことを**MDI(Multi Document Interface)形式**と呼びます。MDIはExcelやWordなど、比較的規模の大きいアプリケーションで利用されます。「俺のプログラムにゃこんなおおげさな枠組みは要らないぜ」という方もご心配なく。より単純なアプリケーションのためのスケルトンを作成する方法も、次節でちゃんと紹介します。

ここはとにかく、Visual C++が作り上げた、このMDI形式のプログラムで遊んでみましょう。まず[表示]-[ツールバー]を選択すると、ツールバーが消えたり、再び表示されたりします。[表示]-[ステータスバー]ならステータスバーが消えたり表示されたりします。[ウィンドウ]-[新しいウィンドウを開く]コマンドを実行すると、新しいウィンドウが開きます。

このように、AppWizardが生成したスケルトンには、一般のWindowsアプリケーションと同様な多くの機能が最初から組み込まれています。とくに、今あげたような定型の処

*1 メニューから[ビルド]-[実行]を選択した場合、あるいはツールバーの<プログラムの実行>ボタン(ビックリマークのボタン)をクリックした場合は、デバッグ情報が読み込まれないため、起動が早くなる。

理については、プログラマが手を入れる必要さえありません。

しかし、スケルトンが提供してくれない機能も当然あります。今度は[ファイル]－[開く]コマンドを実行してみましょう。すると**図2-8**のような[開く]ダイアログボックスが画面に表示されます。

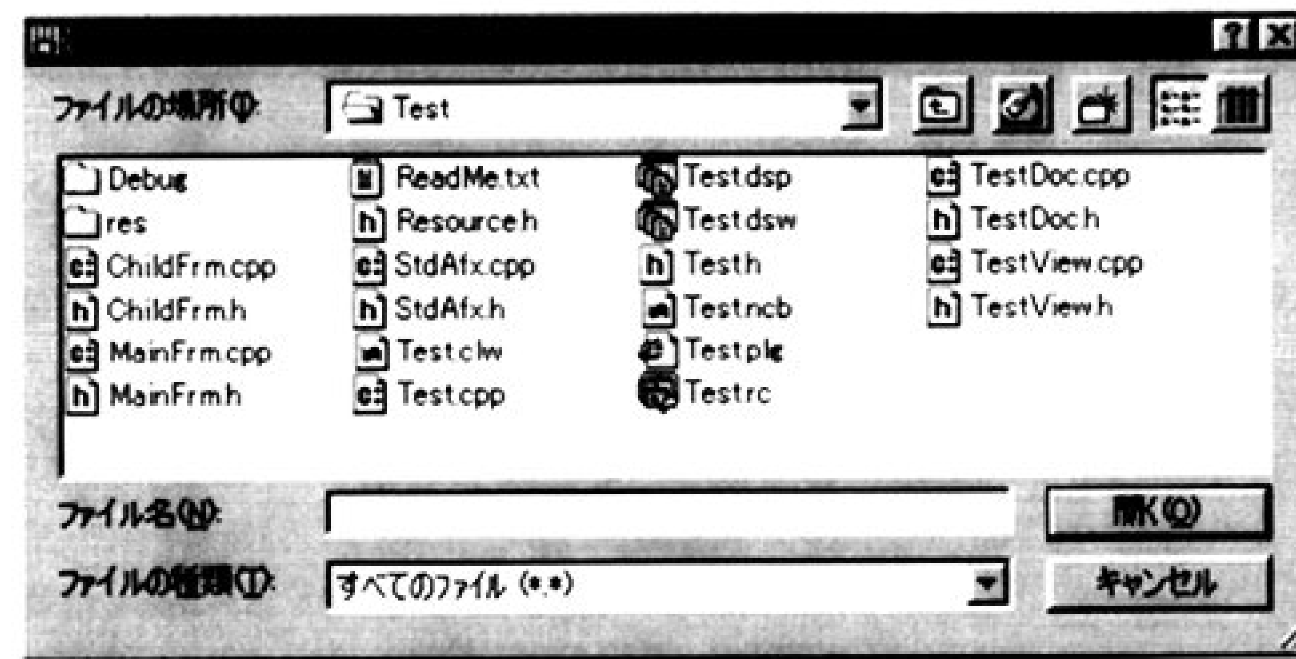


図2-8 [開く]ダイアログ

このダイアログボックスでファイルを指定して<開く>ボタンをクリックしてみてください。残念なことに、ファイルの内容は表示されませんでしたね。なぜなら、ファイルデータの扱い方はプログラムによって異なるからです。テキストを表示するのか、16進ダンプを行うのか、グラフィック図形として扱うのか、こういったことは個々のアプリケーションの動作に依存するわけで、スケルトンの段階では決められません。

AppWizardが作り残したこの余白を埋めて、アプリケーションを完成させること、それがVisual C++によるプログラミングなのです。

2.2 プログラミングの流れをつかもう

前節では、AppWizardにスケルトンを作らせて、それをコンパイルするまでの手順を紹介しました。そこで今度はリソースエディタやClassWizardも動員した、もう少し本格的なプログラミングに挑戦してみようと思います。

一般に、Visual C++を使ってWindowsアプリケーションを作るには、次のようなステップを踏むことになります。

1. **プロジェクトの設計**：プログラムの動作とウィンドウの外見を決める
2. **スケルトンの作成**：AppWizardを実行する
3. **リソースの編集**：リソースエディタを使う
4. **プログラムコードの記述**：ClassWizard(とエディタ)を実行する
5. **コンパイル(デバッグモード)**：デバッグ情報付きの実行ファイルを作る

6. **実行とデバッグ**：失敗なら3.または4.に戻る
7. **コンパイル（リリースモード）**：完成バージョンの実行ファイルを作る

以下の項では、この手順にそって、Visual C++の典型的なプログラミング手法を確認していきましょう。

● プロジェクトの設計とスケルトンの作成

ここで作成するプログラムは、2種類の方法で"Hello World!"というメッセージを表示します。1つ目の方法は直接ウィンドウに文字列を表示するもので、もう1つの方法は[表示]－[ハロー]というメニューを作成し、このメニューを選択するとメッセージボックスを表示するというものです。

前節のスケルトンはMDIアプリケーションでしたが、今回はVisual C++で指定可能なもう1つのアプリケーション形式、**SDI（Single Document Interface）形式**を使いましょう。SDIアプリケーションはウィンドウを1つしか表示しません。また今回のプログラムでは、ツールバーとステータスバーの表示や、印刷関係のコマンドなど、必要ない機能もすべて取り除いてしまうことにします（**図2-9**）。

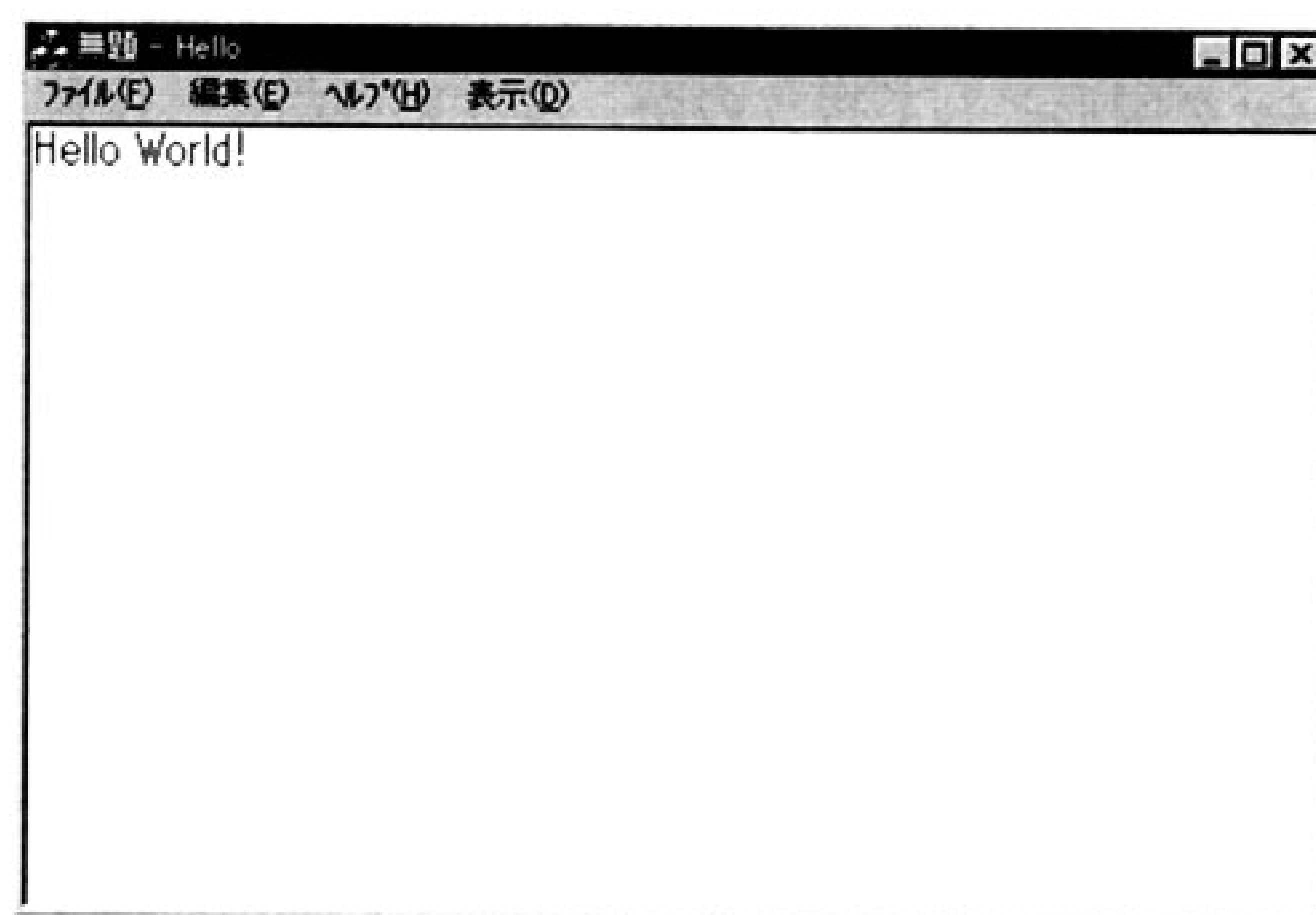


図2-9 プログラムのウィンドウ構成（SDI、ツールバーなし、ステータスバーなし）

まずはスケルトンの作成から始めましょう。さきほどと同じ要領で[ファイル]－[新規作成]を選んで新規作成ダイアログボックスを開き、[MFC AppWizard（exe）]を選択してください。またプロジェクト名は[Hello]とします（**図2-10**）。

さてここからが今回のポイントとなるところです。さきほどはAppWizardのなすがまま、<次へ>ボタンを押し続けただけでソースファイルの雛型を作成しましたが、今回はいくつか変更を加えます。**図2-11**のようにステップ1でラジオボタンの選択位置を変更し、ステップ4で3つのチェックボックスをクリアしてください。ラジオボタンやチェックボックスを変更すると、瞬時に左側に表示されているアプリケーションの完成形のイメー

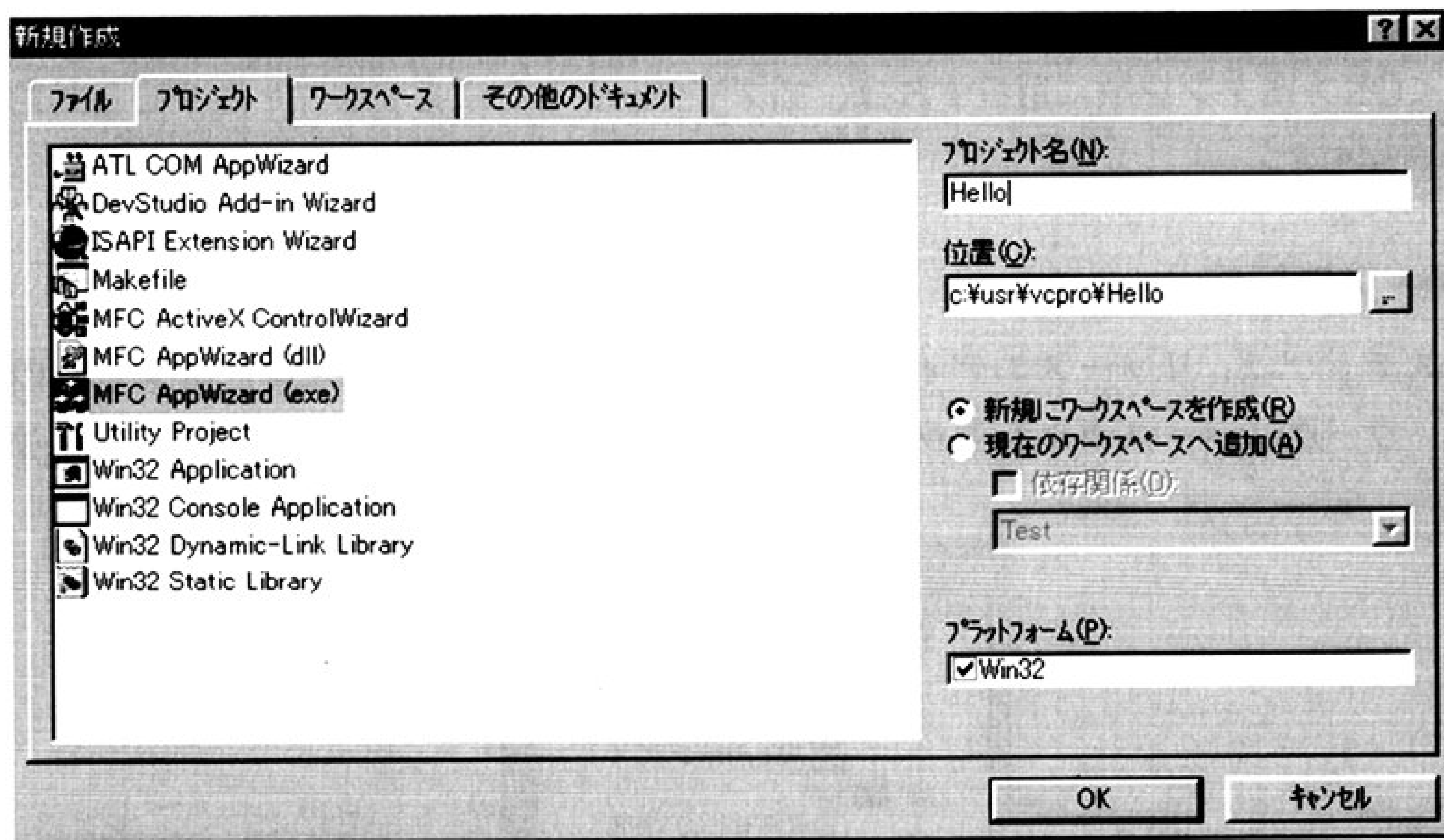


図 2-10 ［新規作成］ダイアログボックス

ジが変わるので、でき上がり具合を容易にイメージすることができます。

SDI アプリケーションを生成する場合は、ステップ 1 で［MDI］ボタンではなく［SDI］ボタンを選択します。また、［ドッキングツールバー］、［初期ステータスバー］、［印刷および印刷プレビュー］は今回は必要ないので、ステップ 4 でこれらのチェックボックスをクリアします。

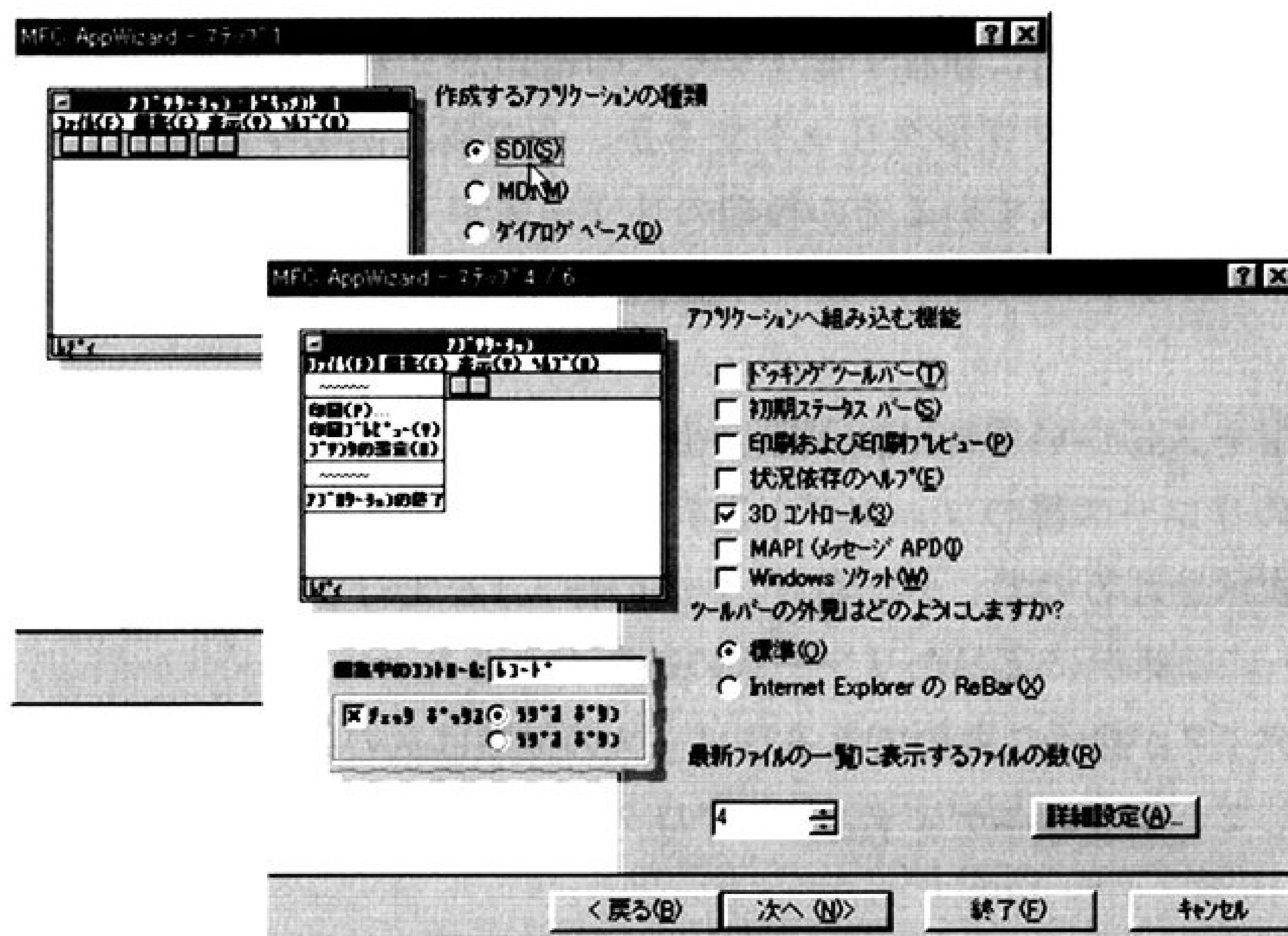


図 2-11 オプションを設定する

以上の操作を行ったら、あとはさきほどと同じようにAppWizardの＜終了＞ボタン、そして＜OK＞ボタンを順に選択すれば、アプリケーションに必要なファイルを自動的に生成してくれます。

● リソースの編集

次のステップは、リソースエディタを使ったリソースの編集作業です。リソースの編集作業は、ワークスペースウィンドウの[ResourceView]ページから始めます(**図2-12**)。

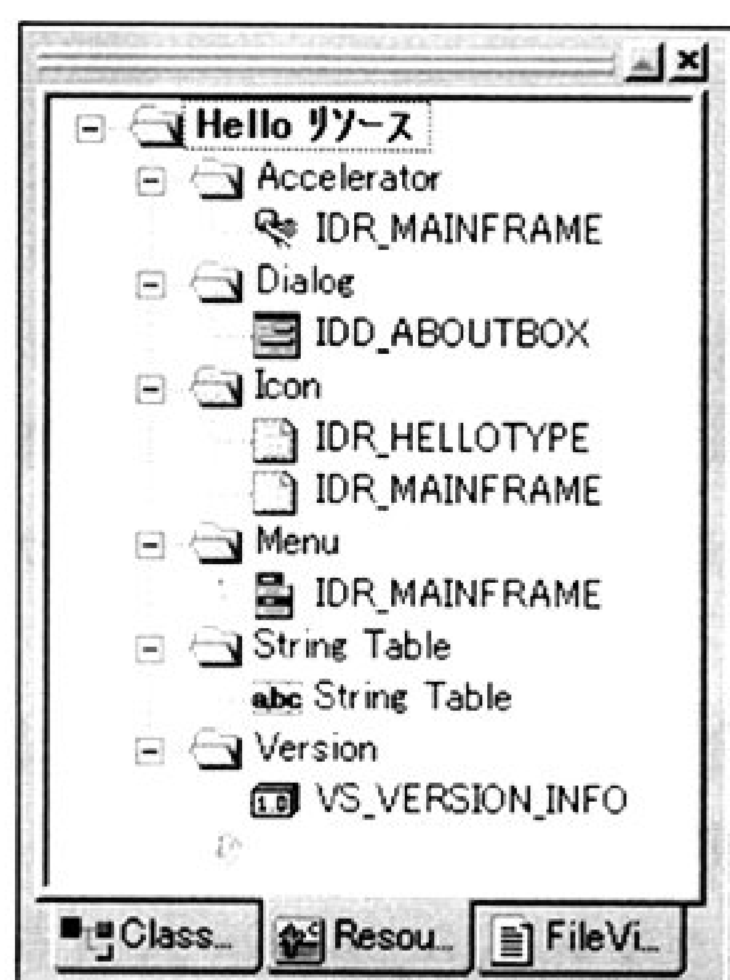


図2-12　ワークスペースウィンドウの[ResourceView]ページ

[Helloリソース]というフォルダをダブルクリックするか、その隣にある+記号をクリックすれば、プロジェクト内に存在するリソースの種類がフォルダとして表示されます。さらに、望みのフォルダをダブルクリックするか、その隣にある+記号をクリックすれば、プロジェクトに組み込まれているその種類のリソースが一覧表示されます。すでにあるリソースをダブルクリックすれば、リソースの種類に応じたエディタが適宜起動されます(**図2-13**)。

基本的なリソースについてはAppWizardが用意をしてくれます。また新規に作成したければ、目的のリソース種のフォルダを選択し、右クリックします。するとショートカットメニューが表示されるので、そこから[……の挿入]を実行すると、新しくリソースが作られます(……には状況に応じたリソースの種類が入る)。

リソースエディタが扱うことのできるリソースの種類は次のとおりです。編集するリソースの種類に応じて、個々のエディタ(ダイアログエディタやメニューエディタなど)が起動します。

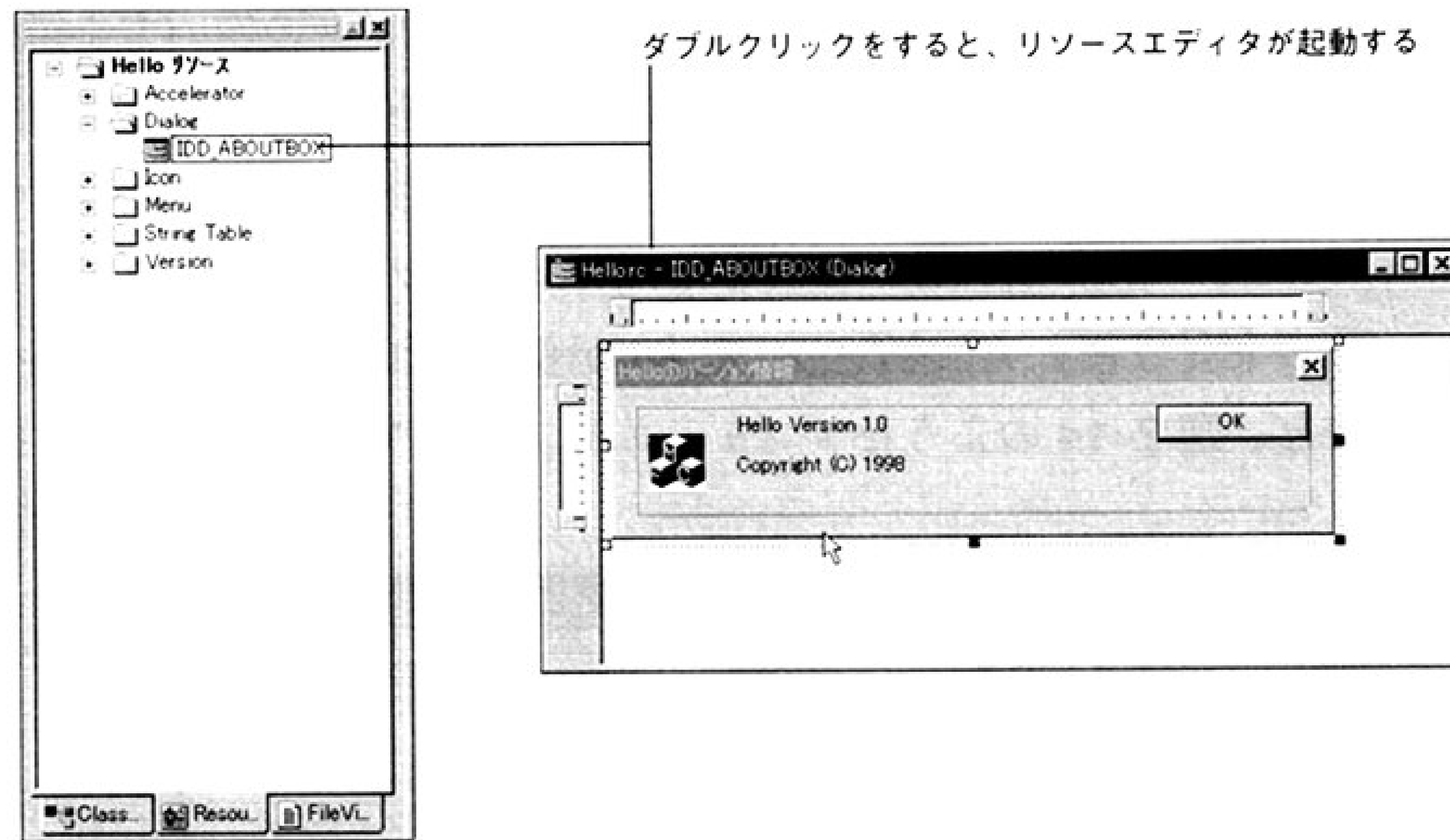


図 2-13　リソースエディタの起動

- **ダイアログボックス**
- **メニュー**
- **ビットマップ**
- **アイコン**
- **カーソル**
- **アクセラレータテーブル**
- **ストリングテーブル**
- **ツールバー**
- **その他のリソース**

プログラム中で、これらのリソースを識別するには**リソース ID** という数値を使います。ただ、実際にはプログラム内でどの数値がどのリソースを示しているのかをわかりやすくするために、それぞれの数値にはそれがどのリソースを示すのかわかるような名前を付けて利用するのが一般的です。本書ではこの数値と名前のことをまとめてリソース ID と呼ぶことにします。

また、リソース ID のうち、ユーザーの操作を受け付けるものに付けられている ID のことをとくに**オブジェクト ID** と呼びます。具体的にはメニュー項目に付けられたリソース ID や画面に表示されるボタンに付けられたリソース ID がこれにあたります（それぞれ**メニュー ID** とか**コントロール ID** と呼ばれることもある）。

このプログラムでは、リソースエディタを使った仕事はただ 1 つ、メニューに［表示］－［ハロー］という項目を追加することです。他のリソースは変更の必要はありません。

リソースエディタには非常にたくさんの機能があり、ここですべてを説明することはできません。今はとりあえず、以下の手順に従ってメニューを変更してください。第 2 部お

よび第3部では、必要に応じてもう少し詳しくリソースエディタの使い方を説明します。

では、ワークスペースウィンドウの[ResourceView]タブをクリックしてください。次に表示されたリソース一覧の中から[Hello リソース]-[Menu]フォルダをダブルクリックすると、そのフォルダの下にAppWizardが生成したメニューリソース[IDR_MAINFRAME](このリソースIDの名付け親もAppWizardです)が表示されます。このリソースIDをダブルクリックすると、**図2-14**のようなメニューエディタのウィンドウが開きます。

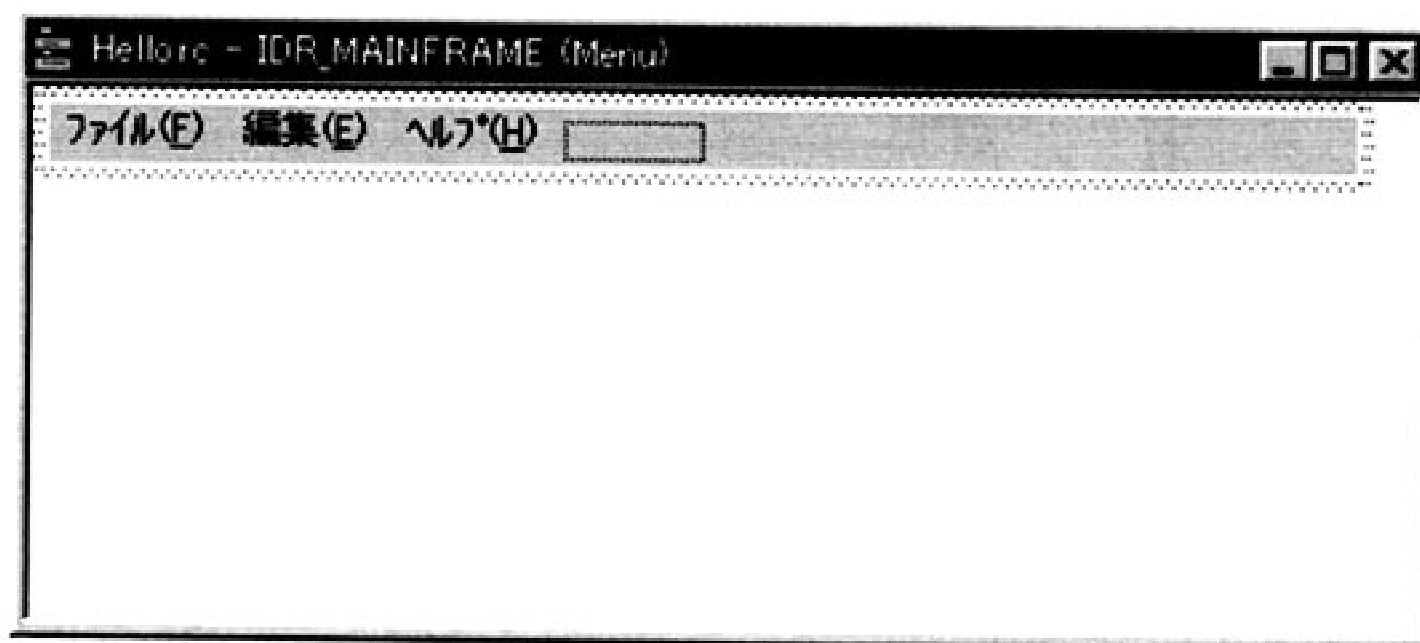


図2-14 メニューエディタ

画面に表示されたメニューの右端にある空白の長方形をダブルクリックすると、プロパティボックスが表示されます。プロパティボックスが表示されたら、[キャプション]ボックスに[表示(&D)]と入力してください(**図2-15**)。ここでは[ID]ボックスには何も入力する必要はありません。

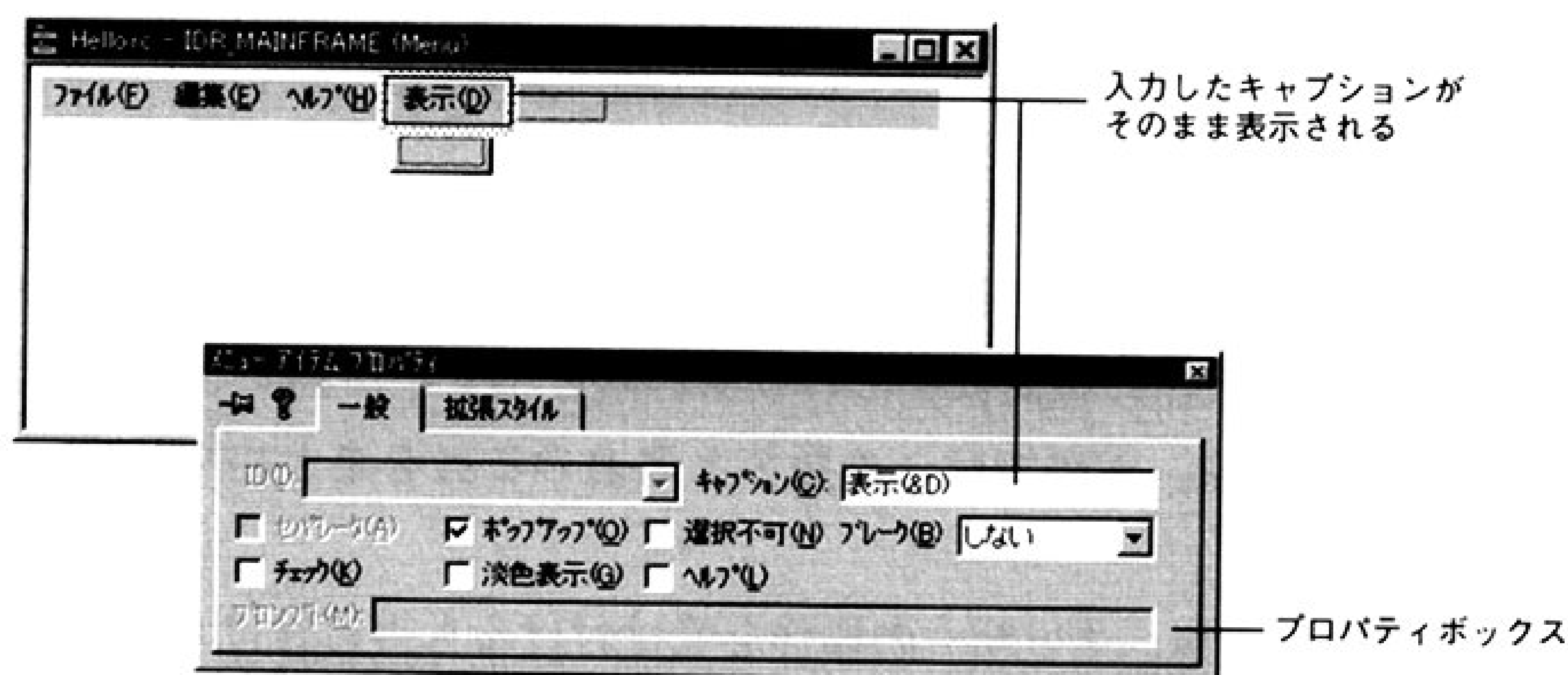


図2-15 プロパティボックス(その1)

メニューバーに[表示(D)]と表示されたら、その下の空白の長方形をダブルクリックします。そして今度はプロパティボックスの[キャプション]ボックスに[ハロー(&H)]と、[ID]ボックスには[ID_CMD_HELLO]と入力します(**図2-16**)。これでメニューに新しい項目が追加されました。



図 2-16 プロパティボックス(その2)

● プログラムコードの記述——ClassWizard

次はいよいよプログラムコードの記述です。ここまで Visual C++のプログラミングとはスケルトンに手を加えることだと述べてきましたが、ならば具体的にどのソースファイルに、どんなコードを書けばよいのでしょうか?

この答は、3章以降の説明を読み、Visual C++のプログラムの構造を理解すれば、おのずとわかってくるでしょう。ここでは、結論からいえば、AppWizard が生成した HelloView.cpp ファイルの中に**リスト 2-1** のような2つの関数を記述するだけです。

リスト 2-1 追加するすべてのプログラムコード

```
// メッセージボックスを利用して文字を表示する
void CHelloView::OnCmdHello()
{
  AfxMessageBox("Hello World!");
}

// ウィンドウの画面に文字を書く
void CHelloView::OnDraw(CDC* pDC)
{
  CHelloDoc* pDoc = GetDocument();

  pDC->TextOut(0, 0, "Hello World!");
}
```

C++言語特有の「::」演算子などにギョッとした方も多いと思いますが、プログラムの内容に立ち入ることはあとの章にまかせ、ここではコード入力の作業に話を絞ります。

2つの関数は、一見似たような関数ですが、コーディングの手順は異なります。まずは CHelloView::OnCmdHello 関数から説明しましょう。この関数を記述するには、Class Wizard の力が必要です。

Developer Studioのメニューから[表示]－[ClassWizard]を選択すると、ClassWizardが起動し、**図2-17**のようなダイアログボックスが開きます。

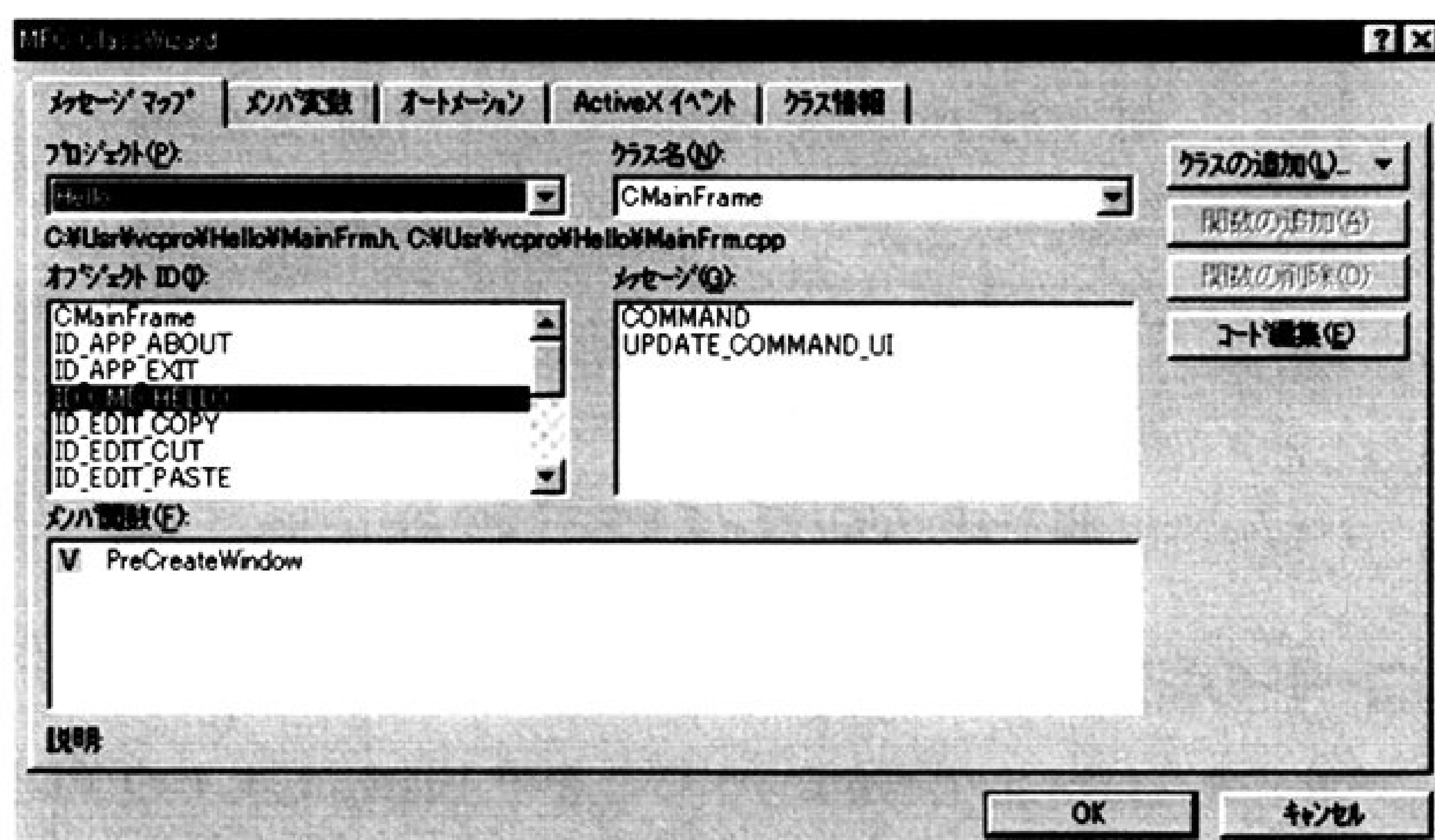


図2-17　ClassWizard

ClassWizardは[メッセージマップ]、[メンバ変数]、[オートメーション]、[ActiveXイベント]、[クラス情報]の5つのページで構成されており、各ページではそれぞれ、メッセージハンドラの作成と削除(後述)、ダイアログボックスに配置したコントロールに対応する変数の追加と削除(第2部を参照のこと)、オートメーション機能の設定、ActiveXイベントの定義、クラス情報の参照と設定といったことを行うことができます。ただし、本書で扱うのは、このうち[メッセージマップ]ページと[メンバ変数]ページだけです。他のページについての詳細はオンラインマニュアル等を参照してください。

ここでは、[メッセージマップ]ページで次の4つのデータを指定します。データはかならずこの順番で指定してください。

- **プロジェクト：Hello**

  [プロジェクト]コンボボックスでは[Hello]が選択されていると思いますが、これはそのままでかまいません(実際のところは変更しようがない)。

- **クラス名：CHelloView**

  [クラス名]コンボボックスを操作して[CHelloView]を選択します。

- **オブジェクトID：ID_CMD_HELLO**

  [オブジェクトID]リストボックスから[ID_CMD_HELLO]というIDを探し、マウスでクリックします。

- **メッセージ：COMMAND**

  [メッセージ]リストボックスから[COMMAND]をマウスでクリックします。

● **関数名：OnCmdHello**

ClassWizardの＜関数の追加＞ボタンをクリックすると、関数名入力のためのダイアログボックスが表示されるので、そこで［OnCmdHello］という名前を指定します（**図2-18**）。

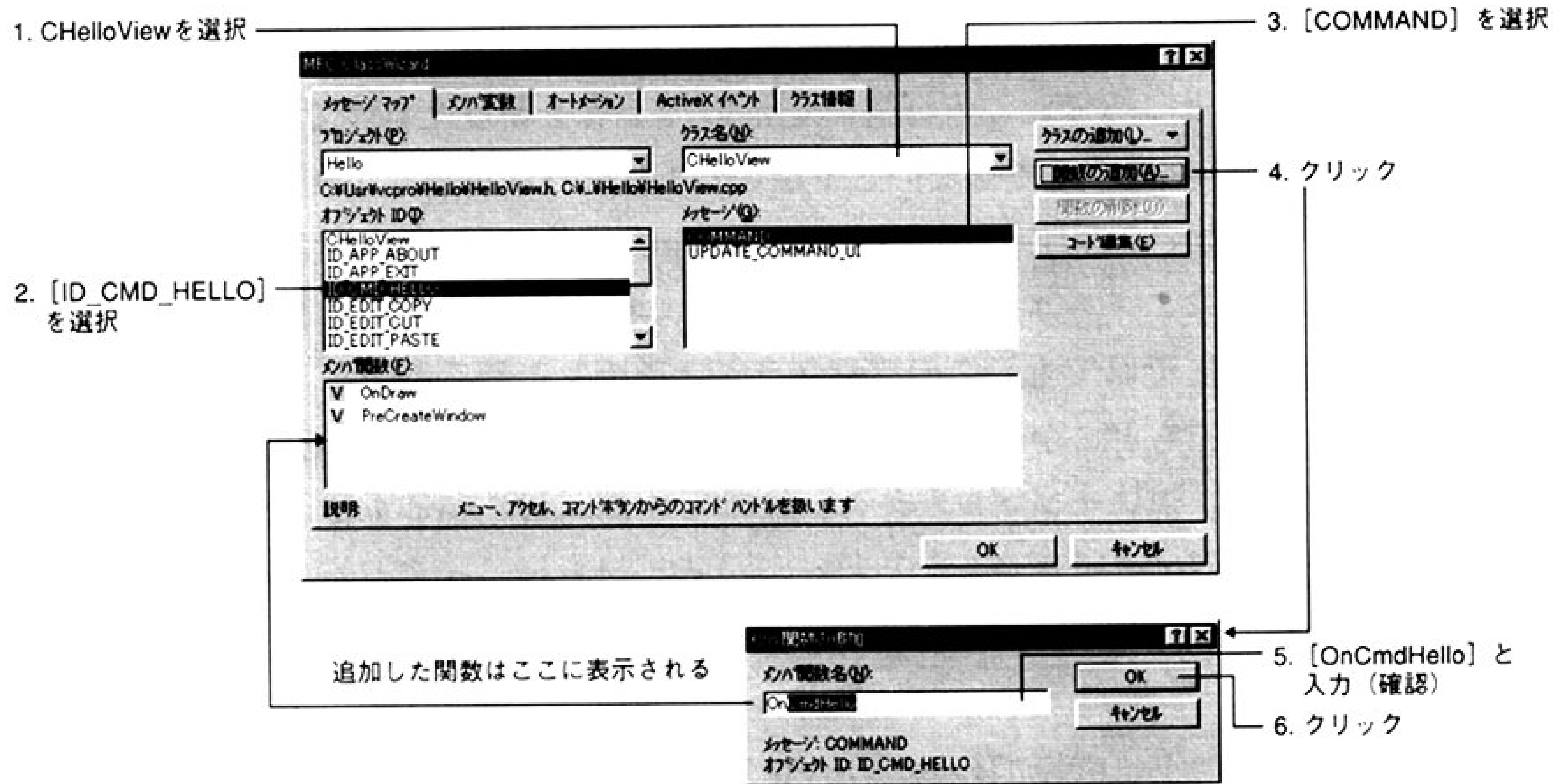


図2-18　ClassWizardを使った関数の作成

そして、［メンバ関数］の中から［OnCmdHello］をマウスで選択し、＜コード編集＞ボタンをクリックすると、エディタが起動してCHelloView::OnCmdHello関数のコードが表示されます。ClassWizardが自動的に関数を作成してくれたのです。ただしこの段階で表示されるコードは関数の枠組みだけで中身はありませんから、次のようにメッセージボックスを表示する1行を書き加えます（**図2-19**）。**図2-19**では、IntelliSenseによるプログ

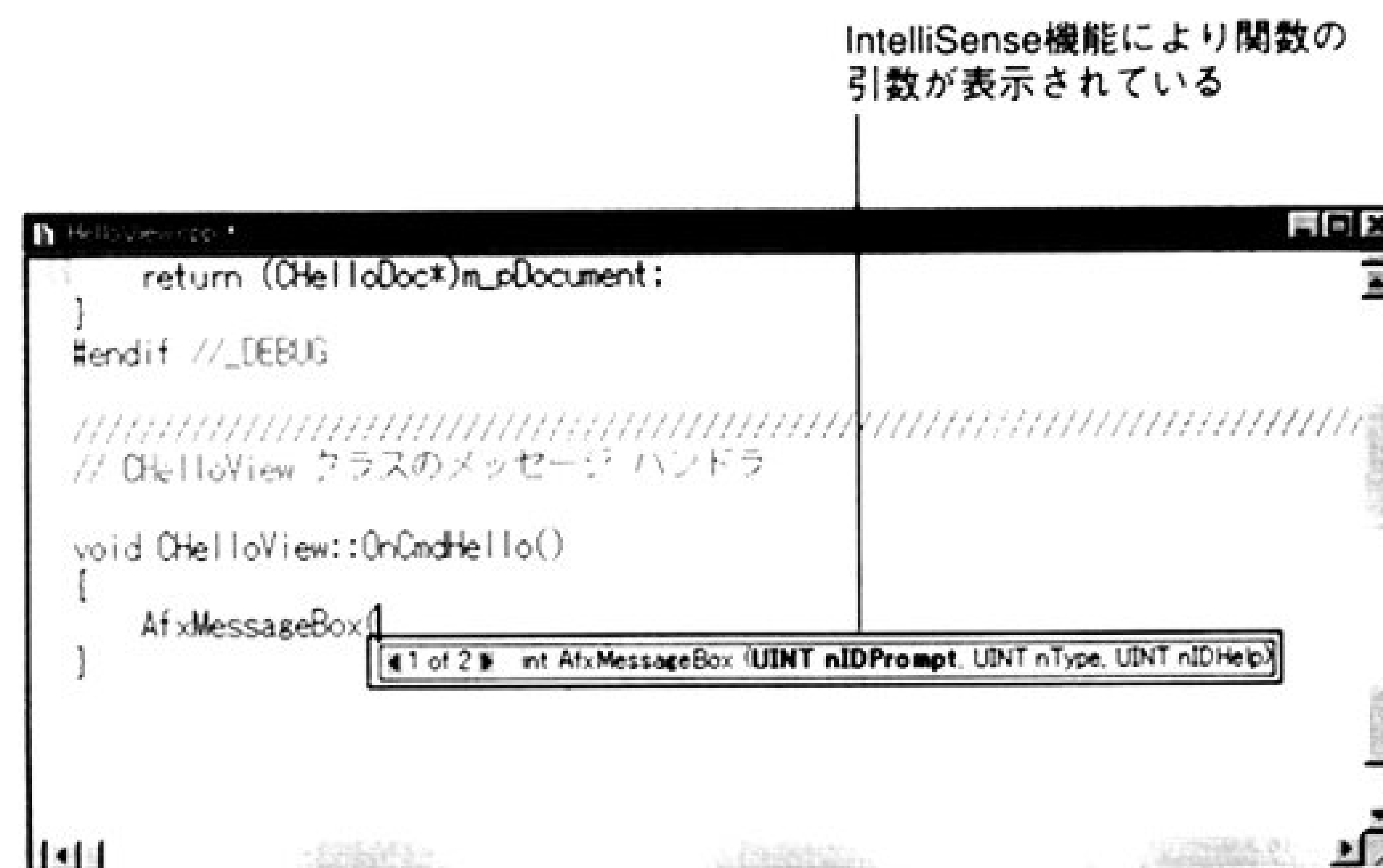


図2-19　プログラムコードの記述

ラマサポート機能の1つして、AfxMessageBox関数の引数がツールチップに表示されていることに注目してください。今後は、関数の引数並びを知るためだけに、HTMLヘルプを参照する回数はかなり減るはずです。

```
AfxMessageBox("Hello World!");
```

以上で、メニューから[表示]-[ハロー]を実行したときに呼び出されるメンバ関数の編集は終わりです。ところで、この作業にClassWizardを使ったわけですが、「たった4行ならClassWizardを使うまでもないじゃないか」と思われたかもしれません。しかし、ちょっと待ってください。今作ったCHelloView::OnCmdHello関数を呼び出す記述はどこにもありません。また、メニューから[表示]-[ハロー]を実行したときに何が起こるかも示されていません。それでも、このプログラムはキチンと動くのです。ということはつまり、この4行以外にもどこかの場所に変更が加えられ、しかもそれはClassWizardの仕業だということです。別にサンプルだから、無理してClassWizardを使ったわけではありません。たったこれだけの編集でもClassWizardがなければ一仕事だから使っているのです。このClassWizardの影の仕事についてはまたあとで解説することにして、次に進みましょう。

次に、ウィンドウの上に直接"Hello World!"と表示するCHelloView::OnDraw関数のコーディング方法を説明します。こちらはClassWizardを使いません。

編集するファイルを開くには、**図2-20**のようにワークスペースウィンドウの[FileView]タブをクリックして、ファイル一覧から対象となるファイル(たとえば、HelloView.cpp)をダブルクリックするか、または、[ファイル]-[開く]を選択して、[ファイルを開く]ダイアログボックスでファイルを選択します。しかし、ここではさきほどCHelloView::OnCmdHello

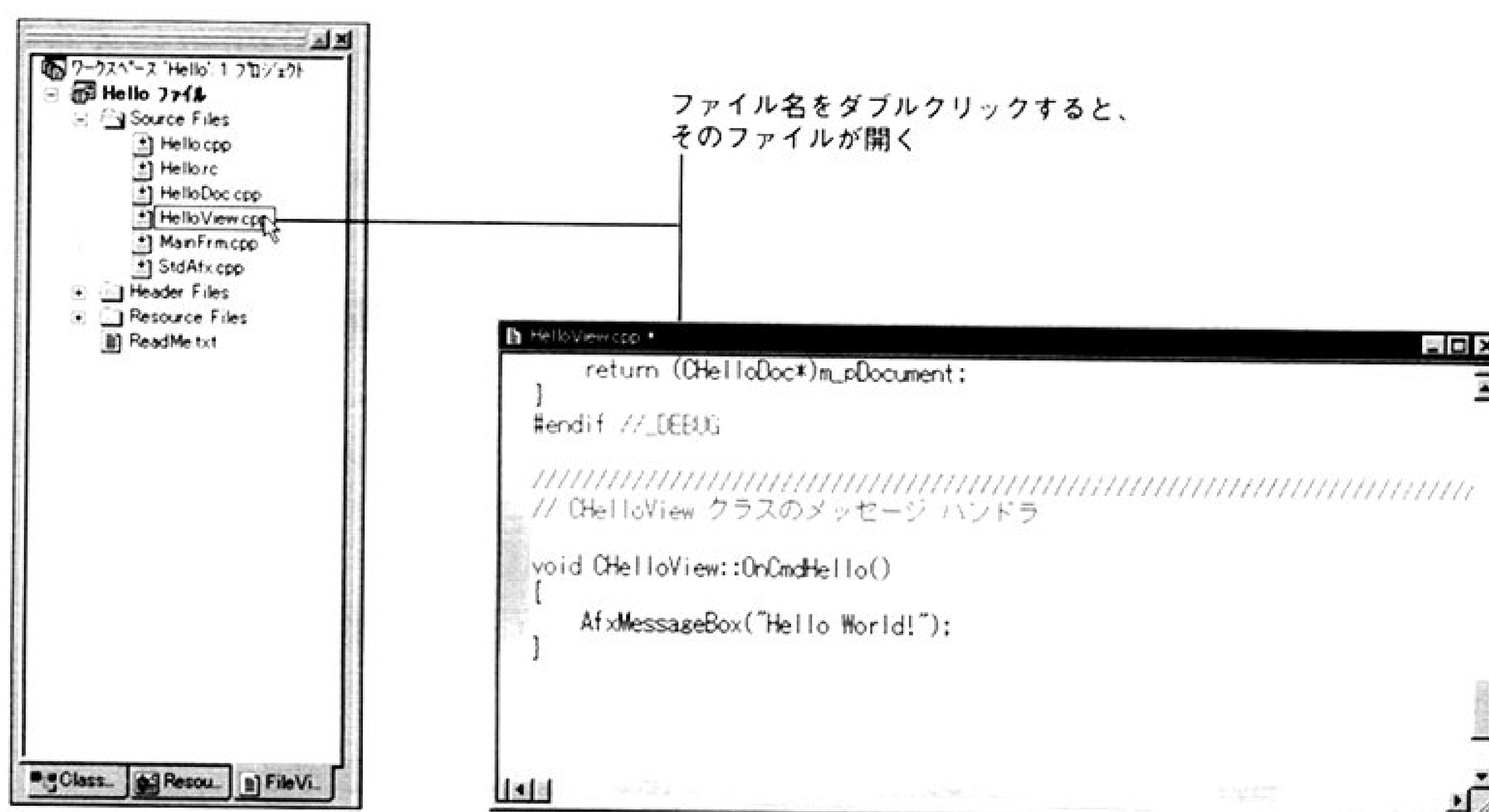


図2-20 HelloView.cppファイルをオープンする

関数を記述した時点でHelloView.cppが開かれているので、ファイルを開く操作はとくに必要ありません。

では、スクロールバーを操作して、次のような部分を探してください。

```
void CHelloView::OnDraw(CDC* pDC)
{
  CHelloDoc* pDoc = GetDocument();
  ASSERT_VALID(pDoc);

  // TODO: この場所にネイティブ データ用の描画コードを追加します。
}
```

これはAppWizardが自動的に生成したスケルトンの一部で、CHelloViewクラスのOnDrawメンバ関数(CHelloView::OnDraw関数)を定義しています。今回はここに以下のプログラムコードを記述することにします。

```
pDC->TextOut(0, 0, "Hello World!");
```

これを書き加えたものが**図2-21**です。たった1行ですが、今回のアプリケーションではこれで十分です。最後に変更したファイルを保存するのを忘れずに行いましょう。[ファイル]-[上書き保存]を選択すればもとのファイル名で保存されます。



図2-21 コードを追加したところ

このように、Visual C++でのプログラムコードの記述方法にはClassWizardを利用するものとプログラマが手作業で行う必要があるものの2種類があることを覚えておいてください。

なお、ファイルを開いたり、関数を編集するのにワークスペースウィンドウの[ClassView]ページを利用することもできます。たとえば、さきほどのCHelloView::OnDrawメンバ関数を編集するには、[ClassView]タブをクリックしたのちに、[Helloクラス]フォルダをダブルクリックして、プロジェクトに存在するすべてのクラスを一覧表示します。ここで[CHelloView]クラスの左にある+記号をクリックしてください。すると、その下にそのクラスのメンバが一覧表示されるので、ここで[OnDraw(CDC* pDC)]をダブルクリック

すれば、ウィンドウにCHelloView::OnDraw関数が表示されます（**図2-22**）。ちなみに、クラス名の部分をダブルクリックした場合にはクラスの定義ファイル（一般には.hファイル）がウィンドウに開かれます。このようにClassViewを使えばクラスのメンバに簡単にアクセスすることができます。

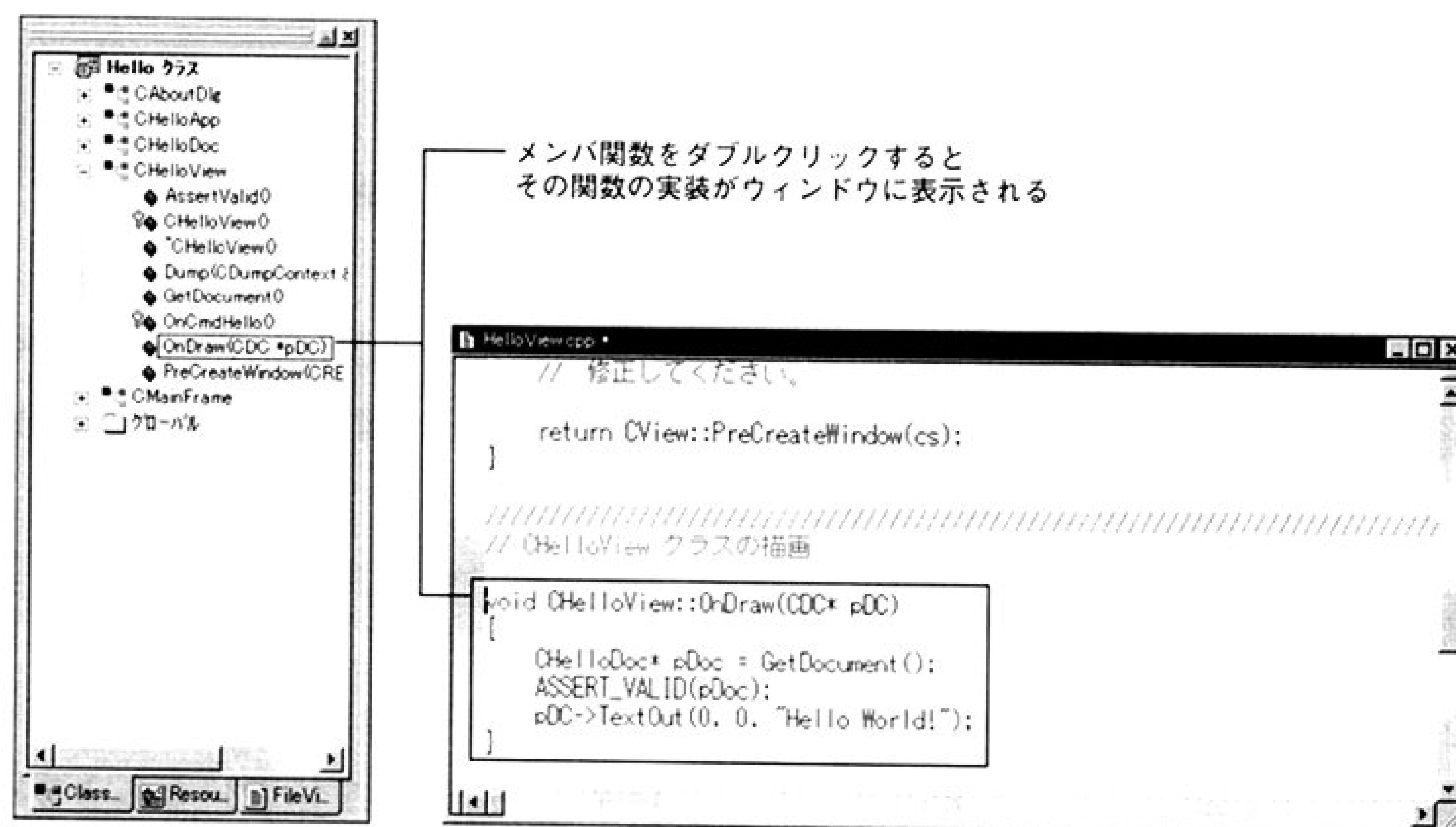


図2-22 ［ClassView］ページを利用した関数の編集

● コンパイル／実行

さて、さきほどはこのままコンパイルを行いましたが、今回は少し手を加えてコンパイルしてみましょう。

まず、メニューから［ビルド］－［アクティブな構成の設定］を選んでください。このダイアログボックスではコンパイル時に使うプロジェクトの構成を指定します。今回、ここではプロジェクトの構成を今まで使っていたデバッグ用の［Hello - Win32 Debug］からリリース用の［Hello - Win32 Release］にしてみましょう。選択をしたら<OK>ボタンをクリックして［アクティブなプロジェクト構成の設定］ダイアログボックスを閉じてください。

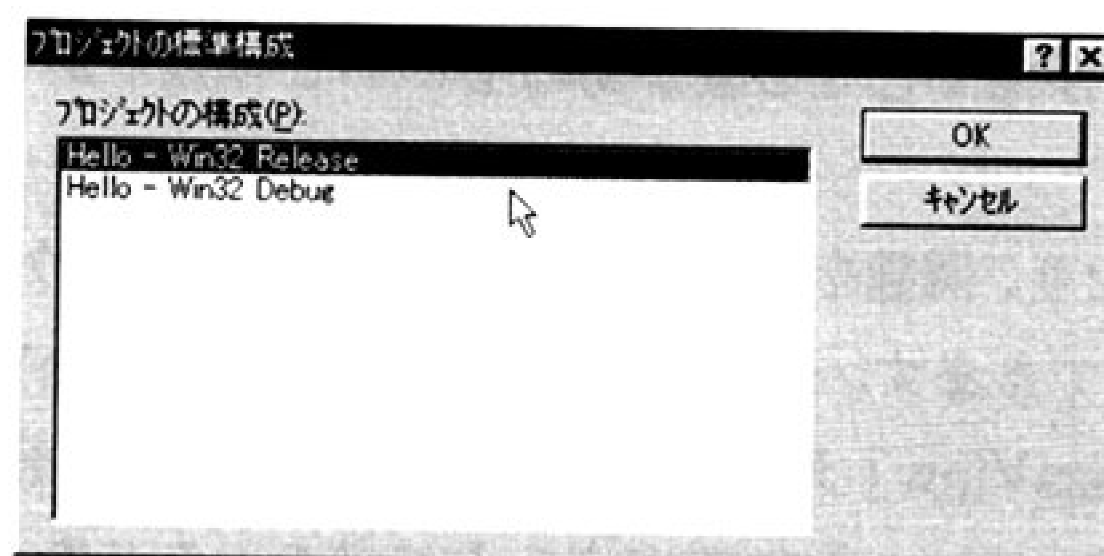


図2-23 プロジェクトの構成の変更

プロジェクトの構成とは、さまざまな環境設定を詰め込んだものです。たとえば、コンパイル時にデバッグ情報を実行ファイルに埋め込むか、最適化を行うか、ブラウザ情報ファイルを生成するか、などの設定がこれに含まれます。これらの設定をプロジェクトの構成という形で複数用意しておき、用途に応じて切り替えれば、膨大な数の環境設定をいちいち何か所もいじくりまわさずにすみます。これがAppWizardが用意してくれた2種類のプロジェクトの構成の正体です。詳しい環境設定の内容は、［プロジェクト］－［設定］を実行すると見ることができます。

プロジェクトの構成で［Hello - Win32 Debug］を選択すると、Visual C++はデバッグに必要な情報をアプリケーション中に埋め込みます。これとは反対に［Hello - Win32 Release］を選択すると、Visual C++はデバッグ情報を作成しません。その代わり、コンパイル時間が若干短くなるとともにプログラムサイズが小さくなります。使い分けとしては、アプリケーションの開発中は［Hello - Win32 Debug］で作成し、開発終了時に［Hello - Win32 Release］で作成するのが一般的です。

プロジェクトの構成を［Hello - Win32 Release］に変更したら、＜ビルド＞ボタンをクリックしてコンパイルしてみましょう。

どうでしたか？ 最後までコンパイルできましたか？ では、今コンパイルしたプログラムを実行してください（**図2-24**）。



図2-24 プログラムの実行画面

ウィンドウの左上隅に"Hello World!"と表示されましたね。また、メニューから［表示］－［ハロー］を選択するとメッセージボックスが表示されるはずです。AppWizardとリソースエディタ、ClassWizardなど、Visual C++が提供する便利なツールとMFCを使ったプログラミングの最大の利点は、ウィンドウを開いたりするようなプログラムの雑多な部分を書かずに、アプリケーション本来の目的にそった部分だけを書くことができるようになることです。本章のサンプルでも、それは感じ取っていただけたかと思います。

では、プログラマがアプリケーション本来の目的に集中するためにMFCとAppWizardが生成してくれたコードはいったい何なのでしょうか? 3章ではAppWizardが生成したコードの謎を探るために、Visual C++(とMFC)を利用して作成するプログラムの基本的な構造や、MFCの基礎知識などについて説明することにします。

3 プログラムを解剖してみよう

前章では非常に単純なプログラムを例に Visual C++でのプログラミング作業を説明しました。このプログラムで実際に入力したコードは、ウィンドウへの文字出力に1行、メッセージボックス表示のために1行と、ごくわずかなものでした。しかしそれは AppWizard が生成した山のようなコードが背景にあって初めて動作したのです。いったいこの膨大なコードは何をするためのものなのでしょうか？ 本章では、C++言語と Windows アプリケーションに関する知識を解説したあとに、AppWizard が生成したコードがどんなものなのか、その意味を探っていくことにします。また、Visual C++で作成するアプリケーションの生命の源ともいえるクラスライブラリ、MFC についても説明します。

3.1 クラスとメッセージとMFC

プログラムの内容を理解する準備として、まずは Visual C++のプログラムの基本となる、3つの知識を頭に入れてもらいましょう。

1つ目はC++言語の**クラス**の役割です。クラスとは、オブジェクト指向プログラミングのエッセンスを取り入れることで、C 言語の構造体の機能が大幅に拡張されたものですが、ここでは「クラスとはこんなもの」といった感じの概念的な説明だけにとどめてあります。Appendix では、構造体とクラスの比較など、もう少し詳しい説明をしていますから、そちらにもぜひ目を通しておいてください。

2つ目は Windows アプリケーションの動作を支配する**メッセージ**の働きです。Windows アプリケーションがどのようなしくみで動作するのかについて説明をします。

そして、3つ目は Windows アプリケーションを構成する各種部品の集合体、MFC についてです。Visual C++を使いこなすということは、すなわち MFC を使いこなすことにほかなりません。本章ではその外面的な部分と内面的な部分の両方について、基本的な概念を押さえていくことにします。

● クラスとオブジェクト

プログラミングとは、いってみれば現実の世界をコンピュータの上で表現(再現／シミュレート)する試みです。このときに重要なのは、扱う対象をどのようにコンピュータのデータとしてとらえるかということで、それがプログラムの構成にも大きく影響します。

簡単な例としてコンピュータの上で人間を表現することを考えてみましょう。プログラム中で複数の人間を扱うとき、それぞれを別のデータとして表現することもできますが、共通する要素をくくり出した**型**を用意しておくと、いろいろな処理が単純化できます。

たとえば、人は名前／年齢／身長／体重など、個人的な情報を持っています。また人はご飯を食べたり、寝たりしますが、これらの行動は誰でもほとんど同じです。この２つの要素を考え合わせると、人という存在は以下のようなデータの集合として表現できるでしょう。

```
人 {
固有の値 :
  名前 ;
  年齢 ;
  身長 ;
  体重 ;
  ...
動作 :
  ご飯をたべる {体重が増え所持金が減る};
  寝る{布団やベッドがしわくちゃになる};
  ...
}
```

このように、表現する対象をその固有の値(これを**メンバ変数**と呼ぶ)と可能な動作(これを**メンバ関数**と呼ぶ)という２種類の情報によって表したものが、C++言語におけるクラスです。

ただし、クラスはあくまでも概念を記述したものでしかありません。現実世界で生きているのは実体のない人ではなく、たとえば「西村」という個人です。これと同じように、コンピュータのメモリの中にも、人というクラスではなく、クラスを具現化した何かが存在しています。この何かのことを**オブジェクト**といいます。

次のようにすると、人クラスのオブジェクトがコンピュータ上に用意されます。これを**オブジェクトの定義**といいます*1。

```
人　西村 ;
人　桜田 ;
```

こうして人クラスのオブジェクトとして定義された「西村」や「桜田」には人クラスの特

---

*1 オブジェクトの宣言とは違うことに注意。オブジェクトを定義すると、そのオブジェクトにはメモリが割り当てられるが、オブジェクトの宣言はそのオブジェクトがプログラム内のどこかで定義されていることを知らせるだけでメモリを割り当てたりはしない。

徴が与えられ、たとえば名前や年齢などの情報（メンバ変数）を持ち、ご飯を食べる、寝るといった動作（メンバ関数）もすべて行うことができます。

また、人が生まれてくるときには名前や性別などの情報はすでに決まっているように、オブジェクトも生成されるときにそのメンバ変数の初期化を行うことができるようになっています。これを行ってくれるのが**コンストラクタ**と呼ばれる特殊な関数です。コンストラクタの具体的な使い方は第2部以降で説明をしていきますが、ここではコンストラクタによってオブジェクトの初期化が行われるということを覚えておいてください。

次にプログラマというクラスを考えてみましょう。プログラマは「プログラムを書く」という特別な能力を持ちますが、他の動作は多くの面で普通の人と変わりません。この場合、人クラスを少し拡張するだけで、プログラマという新しいクラスを作ることができます。プログラマクラスをさきほどと同様な形で表現してみましょう。

```
プログラマ：人 {
動作：
  プログラムを書く｛そら書けやれ書け｝；
}
```

最初の行の「プログラマ：人」は、プログラマクラスが人クラスを特殊化した形で定義されていることを意味します。このようなクラスの特殊化（もしくは拡張）をクラスの継承といい、もとになったクラス（ここでは人クラス）を基底クラス、新しく作られたクラス（プログラマクラス）のことを派生クラスといいます。

次の例は、このプログラマクラスのオブジェクトの「田口」を定義します。

```
プログラマ　田口；
```

派生クラスは、通常は基底クラスの特徴をそのまま受け継ぎます。たとえば上の「田口」にも名前や年齢があり、ご飯だって食べるでしょう。プログラマクラスの定義の中には、これらの情報は明示的に書かれてはいませんが、これらのことはプログラマが人の一種であるというクラス継承の関係から、自動的に定まるのです。

クラスの継承はいろいろな方法で組み合わせることができます。たとえば“人からプログラマ、プログラマからゲームプログラマ”のように継承を重ねていくことも可能ですし、“殿様とカエルからトノサマガエル”のように複数のクラスから1つのクラスを生み出すこともまた可能です。

また、派生クラスでは、基底クラスから受け継いだ能力を新たに定義し直すことができます。たとえば、人の動作の1つに“寝る”というものがありました。普通に考えれば、これは布団とかベッドに入ることを意味します。しかしプログラマは布団やベッドではなく椅子や机の下で寝るものです。

そこで、プログラマクラスでは、“寝る”という動作の定義を変更して以下のように書くことにします。この定義によって、プログラマの寝方が普通の人と異なることが表せます。

```
プログラマ：人{
動作：
  プログラムを書く{そら書けやれ書け};
  寝る{椅子や机があちこちに移動する};
}
```

このように基底クラスの機能を派生クラスで独自のものに書き換えることを、**オーバーライド**といいます。クラスの継承とオーバーライドによって、1つの基本的なクラスから非常にたくさんの派生クラスを作り出していくことができます。

以上で、C++言語に関する説明はとりあえずおしまいです。本書では、C++言語に関する用語が現れるたびになるべく簡単な解説を加えるようにしていますが、より詳しく、厳密に理解をしたいという人はAppendix Aや、オンラインマニュアルのC++言語チュートリアル、市販のC++言語の解説書などを参照してください。では、次にWindowsアプリケーションはいったいどのようにして動いているのか、その基本動作について説明することにしましょう。

● Windowsアプリケーションはどう動く?

Windowsアプリケーションの動作は**メッセージ**というもので制御されます。たとえばウィンドウの上でマウスをクリックすれば、「マウスボタンが押された」というメッセージと「マウスボタンが離された」というメッセージがウィンドウに送られますし、ウィンドウを移動すれば「ウィンドウが移動した」というメッセージがそのウィンドウに送られます。メッセージを受け取ったウィンドウはそれに対応した処理をすることになります。

例として、第1部で作成したHelloアプリケーションの動作を考えてみましょう。Helloアプリケーションは、

- [表示]-[ハロー]を選択すると"Hello World!"と書かれたダイアログボックスを表示する
- クライアント領域の左上に"Hello World!"と表示する

という動作を行うものでした。

この場合、[表示]-[ハロー]を選択すると、「メニューが選択された」というメッセージがウィンドウに送られます。メッセージを受け取ったウィンドウは適切な関数を実行する——すなわち、**メッセージを処理する**か、もしくはデフォルトの処理をWindowsにまかせます。Helloアプリケーションでは、メッセージによってOnCmdHello関数が呼び出され、画面にメッセージボックスが表示されたわけです(**図3-1**)。メッセージボックスが閉じられるとOnCmdHello関数は終了し、ウィンドウは再びメッセージが来るのを待ちます。ウィンドウ左上に"Hello World!"と表示するのも同様な経路を経ています。ただし、こちらは「ウィンドウを描け」というメッセージによって呼び出されるところが先の例とは

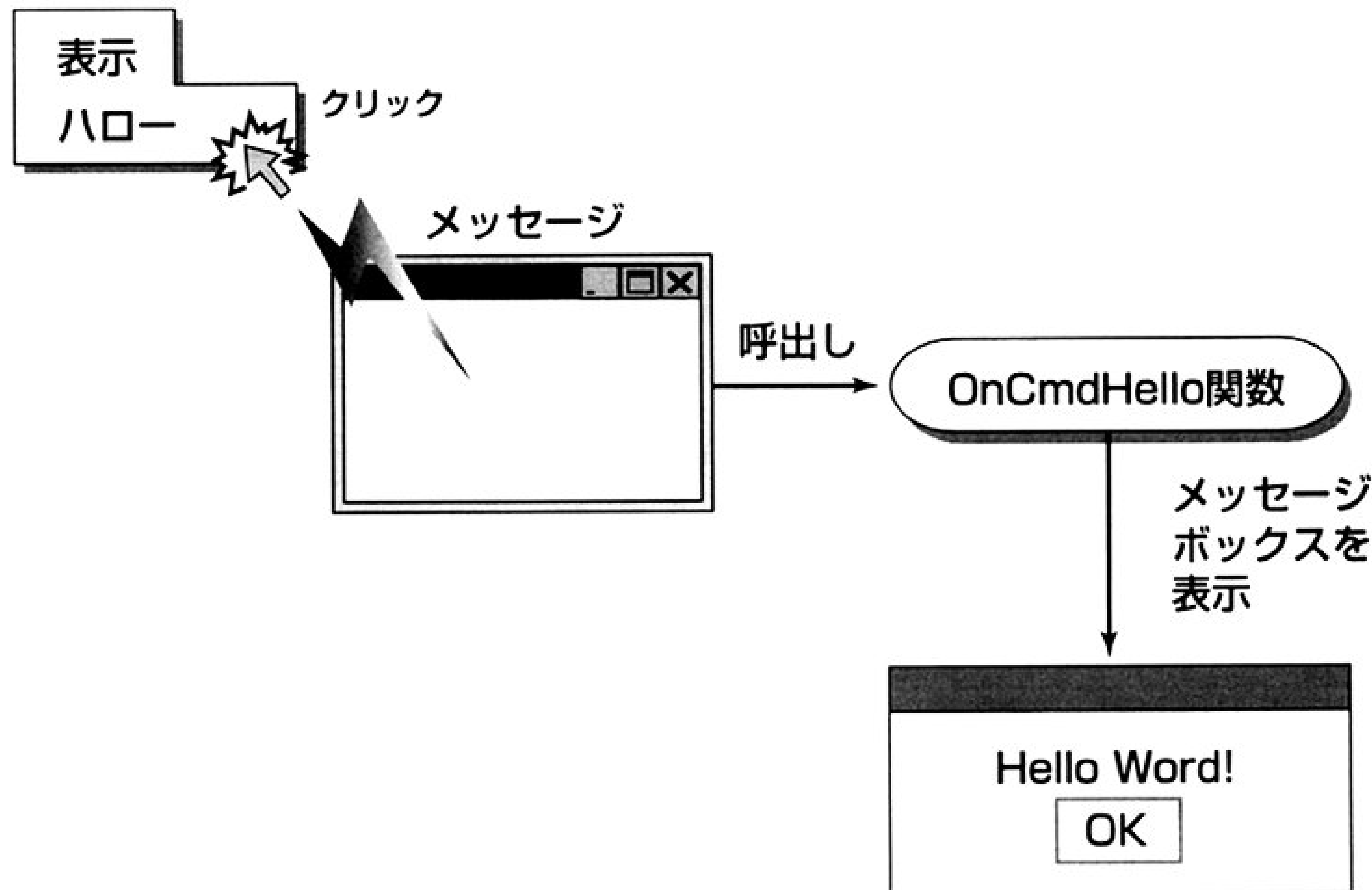


図 3-1 メッセージによるプログラムの実行

異なる点です。

このような特定のメッセージを受け取るたびに実行される関数のことを**メッセージハンドラ**といいます*2。

Windows アプリケーションは、このようにメッセージを受け取り、それを処理し、再びメッセージが送られてくるのを待つという動作を繰り返します。これを**メッセージループ**といい、この点に Windows アプリケーションと MS-DOS アプリケーションの最大の違いがあります。

MS-DOS では、プログラマが定めた main 関数からプログラムの実行が始まり、プログラマが記述した流れにそってプログラムが実行されました。プログラムの中で文字を表示する必要があれば、そこで printf 関数などを実行することによって、MS-DOS のシステムコールが呼び出されて文字出力が行われます。つまり MS-DOS アプリケーションでは、プログラマが記述したコードがシステムを呼び出しながら処理が進んでいくわけです。ところが Windows では、プログラマとシステムの役割が逆転します。Windows アプリケーションでは、プログラマが書くコードはシステムから呼び出されるのです（**図 3-2**）。

いいかえると、Windows の世界ではプログラマはプログラムの流れを取り仕切るのではなく、メッセージを受け取ったときのアプリケーションの動作（という局部的なコード）を記述するということです。これは要するにものの考え方を転換する必要があるということです。「プログラムの流れはこれこれこうだ」的な考え方ではなく「このメッセージをも

*2 このあと説明するが、OnDraw 関数は正確にはメッセージハンドラではない。

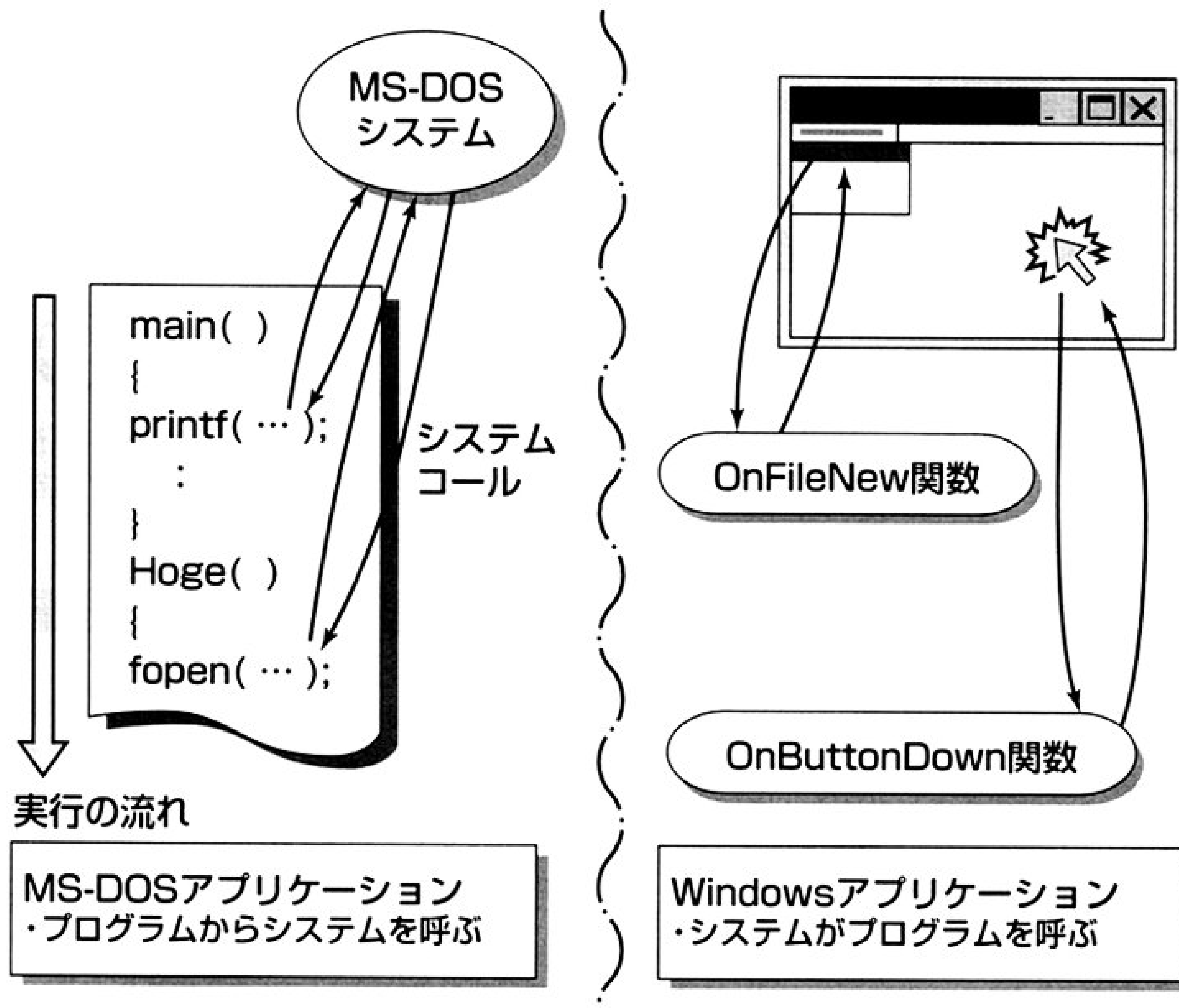


図 3-2　Windows アプリケーションと MS-DOS アプリケーション

らったらこうしよう」とか「こうするには、ここでこのメッセージをもらったときに処理をすればよい」的なものの考え方をしなければなりません。

● Visual C++とWindowsアプリケーション

Visual C++では、画面に表示されているウィンドウやアプリケーションが管理するデータをオブジェクトとして扱います。ユーザーがオブジェクトに対して操作を行うとその操作はメッセージとして操作の対象となったオブジェクトに伝えられます。そして、オブジェクトは送られたメッセージに応じていろいろな処理を行うわけです。Visual C++が提供するクラスライブラリ、**MFC**（**Microsoft Foundation Class**）には、これらのオブジェクトがユーザーの操作（メッセージ）に対してどのような処理をするのかを定義したクラスが数多く存在しています。

MFC はその目的と外見などによって分けられた以下のようなクラスから構成されています。プログラマはすでに用意されているライブラリの中の必要なクラスを利用してプログラムを開発すればよいのです。

- ウィンドウタイトル／境界線／最大化ボタンなどの**ウィンドウの外枠**
- テキストやグラフィックを表示したりユーザーの操作を受け付ける**ウィンドウの内部**

- アプリケーションが管理する**データ**
- ボタンやエディットボックスなどの**コントロール**
- ブラシやペンなどの **GDI オブジェクト**
- **その他**

MFC が提供する多くのクラスには Windows アプリケーションに共通した処理を行うためのコードがすでに用意されています。たとえば、アプリケーションを終了する処理などがそうです。[ファイル]－[終了]を選択すると、「アプリケーションの終了」というメッセージが発生し、その結果、OnAppExit 関数（MFC が用意しているメッセージハンドラの 1 つ）が実行され、ここでアプリケーションは終了します。MFC を使えば、このような定型的な処理はプログラマが記述する必要はありません。

一方、アプリケーションに依存する処理に関してはプログラマがあとでオーバーライドすることを想定して関数が定義されています。たとえば、さきほど使用した OnDraw 関数もその 1 つです*3。OnDraw 関数は画面やプリンタに関する出力を一手に引き受ける関数ですが、MFC が用意した OnDraw 関数は何もしません。その実際の処理はプログラマが関数をオーバーライドして、そこに記述することになります。

ただし、MFC を有効利用するためには、その仕様に従って、プログラムの構造を決定しなければなりません。Visual C++のプログラミングでは、MFC で規定されたいくつかのクラス（とその派生クラス）を、規定どおりの手順で利用する必要があるのです。この結果、作成するプログラムの構造はおのずと決定してきます。このように、プログラムの構造にまで影響を及ぼすライブラリのことを**フレームワーク**と呼びます。

今いったように、MFC を利用してアプリケーションを構築しようとすれば、当然のようにプログラムの基本構造を MFC の仕様に合わせる必要があります。この基本構造の構築を行ってくれるものが、実は AppWizard だったのです。また、Visual C++の作業環境である Developer Studio、ClassWizard やリソースエディタといった各種ツールもフレームワークを十分にサポートして、プログラマの負担を大きく軽減してくれるようになっています。

2 章で記述したコードのうち、画面への文字列出力に関しては MFC が用意した OnDraw 関数を AppWizard が自動的にオーバーライドするように処理をしてくれたので、自分はコードだけを記述すればよかったのです。一方、メッセージボックスを表示するための関数（メッセージハンドラ）は AppWizard が用意してくれなかったので、ClassWizard を使ってプログラマが関数を 1 から作成したというわけです。

*3 OnAppExit 関数／OnDraw 関数などの多くの関数は実際には仮想関数として定義されている。仮想関数に関する詳細は Appendix A を参照のこと。

● MFC の提供する部品

ここでは、MFC がどんな部品を提供し、それらはどのような目的で利用されるかを簡単に説明することにしましょう。MFC にはこんな部品が用意されているんだなと感じてもらえれば、別にこれらのクラスの1つ1つを細かく覚えておく必要はありませんから、軽い気分で読み進めてください。

さきほどもいったように、Windows アプリケーションを構成する部品には、

- **ウィンドウ枠**
- **ウィンドウ内部**
- **アプリケーションが扱うデータ**
- **GDI オブジェクト**
- **各種コントロール**
- **その他**

がありますが、ここでは、この分類にそって MFC が提供するクラスについて説明をしていきます。その前に、MFC 全体がどのような構成になっているかを**図 3-3** に示しておきましょう。**図 3-3** を見ればわかるように、MFC が提供するクラスのほとんどは CObject というクラスをおおもとに派生してきたものです。この中に、ウィンドウ枠の管理を受け持ったり（CFrameWnd クラス）、ウィンドウ内部の管理を受け持ったり（CView クラス）するクラスが数多く含まれています。

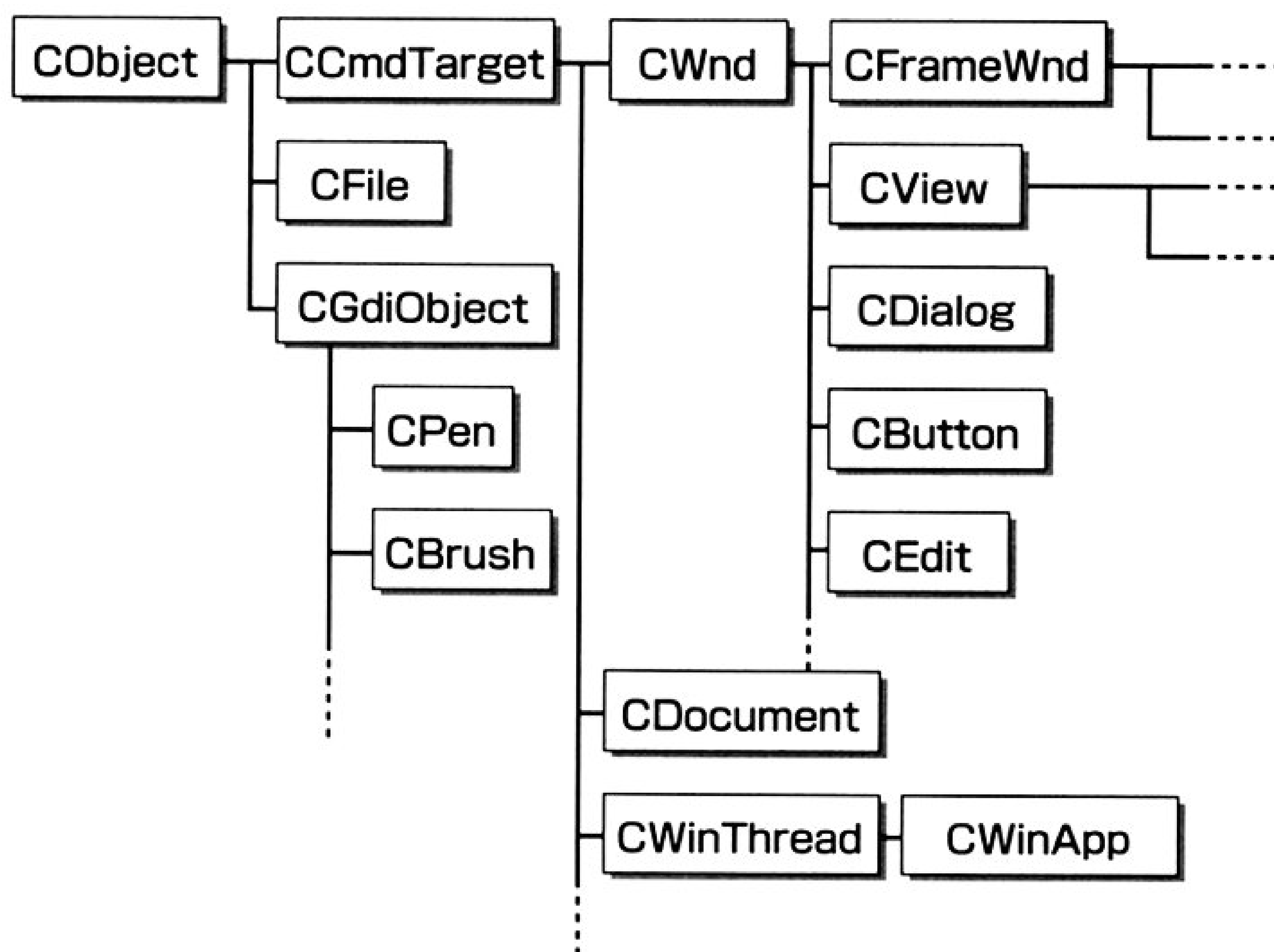


図 3-3 MFC の構造

ウィンドウ枠 —— フレームウィンドウクラス

ほとんどの Windows アプリケーションは、タイトルバー、最大化ボタン／最小化ボタン、境界線などを備えています。これらウィンドウ枠のことをフレームウィンドウといい、MFC はフレームウィンドウを管理するためのクラスとして以下の 5 つのクラスを用意しています。



図 3-4 フレームウィンドウ

- **CFrameWnd クラス**

  CWnd クラスから派生したクラスで、フレームウィンドウを管理します。このクラスのオブジェクトを直接作成することはめったになく、代わりに以下の 4 つの派生クラスを使います。

- **CMDIFrameWnd クラス**

  MDI アプリケーションの一番外のフレームウィンドウ（親ウィンドウのフレームウィンドウ）を管理します。

● **CMDIChildWnd クラス**
MDI アプリケーションの子ウィンドウのフレームウィンドウを管理します。

● **CMiniFrameWnd クラス**
SDI アプリケーションのフレームウィンドウを管理します。

● **COleIPFrameWnd クラス**
OLE 対応アプリケーションを作成するときに使います。

なお、クラス名の先頭に付いている“C”は“Class”を意味するもので、MFC のクラスの名前はすべて“C”で始まっています。みなさんがこれから作成するクラスについてもこの慣習にならって名前付けをすることをお勧めします。

ウィンドウ内部 — ビュークラス

Windows アプリケーションはフレームウィンドウに囲まれたウィンドウ内部（これをクライアント領域という）を使って、ユーザーと対話（やり取り）をします。クライアント領域を管理するためのクラスがビュークラスです。MFC はビュークラスとして以下の 12 のクラスを提供しています。

● **CView クラス**
CWnd クラスから派生したクラスで、CScrollView クラスおよび CCtrlView クラスの基底クラスとなっています。ビュークラスが持つ機能の基本的な部分を提供するのがこのクラスです。

● **CCtrlView クラス**
CView クラスから派生したクラス。ユーザーが直接使うことはほとんどなく、次の 4 つのクラスの基底クラスとしてのみ存在します。

● **CEditView クラス**
CCtrlView クラスから派生したクラス。ビューにテキストエディタとしての機能（テキストの編集／文字列検索／カット&ペーストなど）を簡単に持たせることができます。このクラスに関しては派生クラスを用意することなくそのままプログラムの中で利用可能です。

● **CListView クラス**
CCtrlView クラスから派生したクラス。アイコンを使ったアイテムの一覧を表示することができます。アイテムの表示形式には 4 つのパターンが用意されています。

● **CTreeView クラス**
CCtrlView クラスから派生したクラス。アイコンを使ったアイテムの一覧を階層構造を持たせて表示することができます。



図 3-5 ビュー

● **CRichEditView クラス**

CCtrlView クラスから派生したクラス。CEditView クラスによく似ていますが、あちらがメモ帳と同等の機能を持っているとすれば、こちらはワードパッドと同等の機能を持っているといえます。

● **CScrollView クラス**

CView クラスから派生したクラスで、CView クラスにスクロール機能を追加したものです。

● **CFormView クラス**

CScrollView クラスから派生したクラス。ビューにボタンやエディットボックスなどを利用したユーザーインターフェイスを構築するのに使われます。

● **CDaoRecordView クラス**

CFormView クラスから派生したクラス。DAO データベースのレコードを表示するために使われます。

● **CRecordView クラス**

CFromView クラスから派生したクラス。ODBC データベースのレコードを表示するために使われます。

● **CHtmlView クラス**

Visual C++ 6.0 で新たに追加されたビュークラスです。名前が示すとおり、HTML ファイルを解釈／表示することができます。表示する HTML ファイルは、ローカルのハードディスクにある必要はもちろんありません。必要ならば、インターネットへの接続を行ってくれます。

● **COleDBRecordView クラス**

CFormView クラスから派生したクラスです。OLE DB を使用して、データベースにアクセスする場合に使用します。

GDI オブジェクト

Windows アプリケーションでは、画面へのグラフィックスの出力には GDI オブジェクトというものを利用します。GDI オブジェクトを管理するためのクラスとして MFC では以下のようなクラスが提供されています。実際の GDI オブジェクトの使い方については第 2 部 1 章「GDI はグラフィック表示の合言葉」を参照してください。

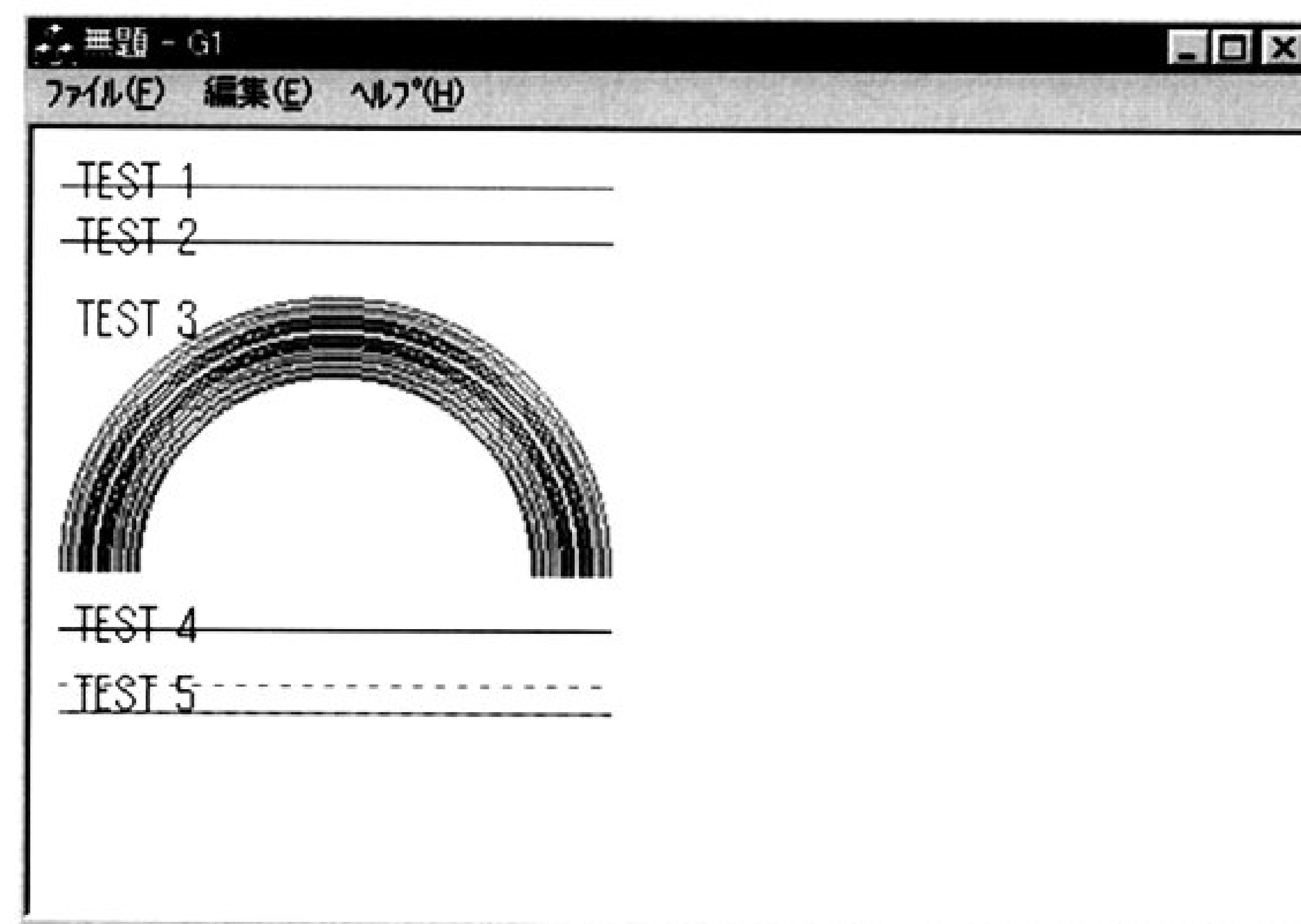


図 3-6　GDI を利用した画面出力

● **CGdiObject クラス**

CObject から派生したクラス。GDI オブジェクトとしての基本的な機能を提供します。また、以下のクラスの基底クラスとなっています。GDI オブジェクトはデバイスコンテキスト（詳細については第 2 部を参照）に描く図形データなどを管理するために使います。

● **CPen クラス**

● **CBrush クラス**

● **CBitmap クラス**

● **CFont クラス**

● **CPalette クラス**

● **CRgn クラス**

コントロールとダイアログボックス

コントロールとは、ダイアログボックスやビューの上に貼り付けられるボタンやエディットボックス、リストボックス、コンボボックスなどのことです。これらの部品は MFC では、以下のようなクラスで管理されています。これらのクラスはすべて CWnd クラスから派生したものです。また、ダイアログボックスは CDialog クラスがその管理を行っています。

ダイアログボックスといくつかのコントロールの実際の利用については第 2 部 3 章「ダイアログボックスを使ってみよう」を参照してください。

● **CDialog クラス**

すべてのダイアログボックスの基本となるクラス。ダイアログボックスを扱うための関数を提供しています。

● **CButton クラス、CBitmapButton クラス**

● **CEdit クラス**

● **CListBox クラス**

● **CComboBox クラス、CComboBoxEx クラス**

● **CStatic クラス**

● **CAnimateCtrl クラス**

● **CListCtrl クラス、CTreeCtrl クラス**

● **CProgressCtrl クラス**

● **CRichEditCtrl クラス**

● **CSliderCtrl クラス**

● **CToolTipCtrl クラス**

● **CDateTimeCtrl クラス**

- **CReBar クラス**
- **CMonthCalCtrl クラス**
- **その他のクラス**

アプリケーションが管理するデータ

アプリケーションが管理するデータには大きく分けると2種類あります。1つはメモリ上にロードされ現在使用されているデータで、もう1つはハードディスクなどの外部記憶装置にデータファイルとして保存されているデータです。MFCでは、これらのデータを扱うために以下のようなクラスを用意しています。これらのクラスを実際に利用するプロジェクトは第3部で紹介します。

- **CDocument クラス**
  CCmdTarget クラスから派生したクラス。アプリケーションが使用するデータを管理するためのクラスで、ビュークラスとも深い関係にあります(詳細については第3部を参照)。CArchive クラスのオブジェクトを介してデータをメモリと二次記憶に保存されたファイルとの間でやり取りすることもできます。
- **CArchive クラス**
  CDocument クラスが管理するデータとファイルの間の入出力を行う際に利用します。
- **CFile クラス**
  CObject から派生した、ファイルを扱うためのクラスです。CArchive クラスのオブジェクトも内部的には CFile クラスのオブジェクトを利用してドキュメントとファイル間のやり取りを行っています。

その他

これまでに説明したクラスの多くは CObject クラス／CCmdTarget クラス／CWnd クラスから派生したものでした。ここでは、これら3つのクラスと CDC クラス／CWinApp クラスについて説明をします。

- **CObject クラス**
  CObject クラスは、MFC の根本に存在するクラスで、MFC が提供するクラスの多くは CObject クラスから派生したものです。このクラスは「シリアライズ」、「ランタイムクラス情報の取得」、「オブジェクトの診断」などの機能を提供します。シリアライズ／ランタイムクラスについては第3部で解説をします。
- **CCmdTarget クラス**
  Windows アプリケーションは、メッセージを処理することで動作することはすでに述べたとおりですが、メッセージを処理することができるクラスは、すべてこの CCmd Target クラスの派生クラスです。このクラスの派生クラスとしては、CWnd クラス

／CDocument クラス／CWinApp クラスなどがあります。4 章で説明するメッセージマップに関するアーキテクチャを提供するのもこのクラスです。

● **CWnd クラス**

CWnd クラスは CCmdTarget クラスから派生したクラスで、画面に表示される多くのウィンドウの基本的な機能を提供します。

● **CDC クラス**

Windows アプリケーションの出力は基本的にはすべてデバイスコンテキスト（DC）というものに対して送られます。DC を扱うためのクラスがこの CDC クラスです。文字を出力したり、図形を描いたりといった作業はすべてこのクラスのオブジェクトを対象に行われます。DC とその操作については第 2 部 1 章「GDI はグラフィックス表示の合言葉」で説明します。

● **CWinApp クラス**

CWinApp クラスは、アプリケーションの実行に必要な初期化／ドキュメントテンプレートの管理などを行うクラスで、MFC を利用するすべてのアプリケーションが、このクラスの派生クラスを 1 つだけ必要とします。

3.2 MFCを使ったWindowsアプリケーションの基本構成

MFC の概要の説明が終わったところで、この節では、Hello プロジェクトを例として、AppWizard が作り出す Windows アプリケーションがどのような MFC のクラスを含み、それぞれがどのような役割を持つのか紹介しましょう。

● プログラムを構成するファイル群

すでに述べたように、Hello プロジェクトは非常に多くのファイルから構成されていますが、その中でもとくに重要なファイルが、以下に示す 4 つのヘッダファイル（インクルードファイル、.h ファイル）と 4 つの実装ファイル（インプリメントファイル、.cpp ファイル）です。

- Hello.h
- MainFrm.h
- HelloView.h
- HelloDoc.h
- Hello.cpp
- MainFrm.cpp

- HelloView.cpp
- HelloDoc.cpp

C 言語ではヘッダファイルを、関数のプロトタイプ宣言や構造体の定義などを記述するために使いました。C＋＋言語ではそれに加えて、クラスの定義もヘッダファイルに記述します。また、ヘッダファイルに対応する実装ファイルにはクラスのメンバ関数の定義を記述します。4 つのヘッダファイル（と対応する 4 つの実装ファイル）があるということから、MFC アプリケーションは 4 つのクラスから構成されていることが予想できます。それでは、その 4 つのクラスとはどんなものなのでしょうか？ それぞれのファイル（の組）では以下のような 4 つのクラスが定義されています*4。

- **Hello.h と Hello.cpp：CHelloApp クラス**
  CWinApp クラスからの派生クラス。アプリケーション全体で利用されるデータを管理します。
- **MainFrm.h と MainFrm.cpp：CMainFrame クラス**
  CFrameWnd クラスからの派生クラス。フレームウィンドウに対する操作を処理します。
- **HelloView.h と HelloView.cpp：CHelloView クラス**
  CView クラスからの派生クラス。クライアント領域に対する操作を処理します。
- **HelloDoc.h と HelloDoc.cpp：CHelloDoc クラス**
  CDocument クラスからの派生クラス。ドキュメントを管理します。

これら 4 つのクラスはすべて MFC が提供するクラスから派生したものです。ユーザーの操作によって発生したメッセージは、メッセージループの中でこれらの 4 つのクラスのオブジェクトのうち適切と思われるものに送られます。メッセージを受け取ったオブジェクトは、対応するメッセージハンドラを実行します。

また以上の 8 つのファイル以外に、プロジェクトの開発を進めるうちに次のファイルもプロジェクトフォルダに作られます（**表 3-1**）。ただしこれらのファイルは基本的に Developer Studio によって編集されるものばかりなので、プログラマが直接編集することはほとんどありません。唯一 Stdafx.h だけがプログラマが編集するファイルです。この中には windows.h などシステムインクルードファイルが#include ディレクティブによってインクルードされています。これらのシステムインクルードファイルは数万行にもおよぶ巨大なファイルなので、毎回コンパイルするとたいへん時間がかかります。そこでファイルが変更されない限り、二度とコンパイルせずに済ます機能が Visual C++に用意されました。これが**プリコンパイルヘッダ**機能です。この機能を利用するためには StdAfx.h を編集して、対象と

*4 実際にはアバウトダイアログボックスを表示するためのクラスの定義も含まれるが、アプリケーションの本質には関係ないのでここでは無視する。

ファイル名	名前	目的
Hello.aps	リソースIDの定義ファイル	リソースIDと数値の対応を定義する。削除しても自動的に作られる
Hello.clw	ClassWizard用作業ファイル	ClassWizardが参照する作業ファイル。削除しても自動的に作られる
StdAfx.cpp	プリコンパイルヘッダ用ソースファイル	プリコンパイルヘッダ機能を利用するためのダミーファイル
StdAfx.h	プリコンパイルヘッダ用ヘッダファイル	めったに変更されることのないインクルードファイルを指定すると、ビルド時間が短縮される
Hello.dsw	Developer Studio Workspaceファイル	プロジェクトを開くためにこのファイルを指定する
Hello.dsp	Developer Studio Projectファイル	従来の.makファイルから変更された。ビルド手順が記述してある
Hello.opt	ワークスペースオプションファイル	.dswファイルのサポート。削除しても自動的に作られる
resource.h	リソースインクルードファイル	Hello.rcから参照されるインクルードファイル
Hello.plg	ビルド結果ファイル	アウトプットウィンドウに出力されるものと同じ。ビルドするたびに作られる
Hello.rc	リソースファイル	プロジェクトで使用されるすべてのリソースが定義されている

表3-1　作業中に作成されるファイル

するインクルードファイルをインクルードします。ただし頻繁に変更されるインクルードファイルまで含めてしまうと逆にビルド時間が延びることがあるので、注意が必要です。

● ウィンドウは見た目が肝心

話を4つのヘッダファイルと実装ファイルに戻しましょう。これらのファイルにそれぞれ1つずつクラスが定義されているのはわかりましたが、なぜ最初から4つものクラスが必要なのでしょうか？ しかも、たかだか決まった文字列を表示するだけのHelloプログラムでさえ、これらのクラスを1つとして省略することはできないのです。この謎を解き明かすために、まずはプログラムの構造をビジュアルな視点から考えてみることにしましょう。

図3-7はWindowsアプリケーションをビジュアルな視点で観察したものです。MDIアプリケーションとSDIアプリケーションでは多少構成が違いますが、基本的にはウィンドウの外観は、**フレームウィンドウクラス**と**ビュークラス**の2種類のクラスで管理されることがわかります。

さきほども説明したように、フレームウィンドウクラスは、ウィンドウ枠やタイトルバー、メニューなどアプリケーションのクライアント領域以外の部分（ノンクライアント領域）を管理するためのクラスで、ビュークラスはアプリケーションのクライアント領域を管理するためのクラスです。その一方で、ビュークラスはマウスやキーボードからのユーザーの

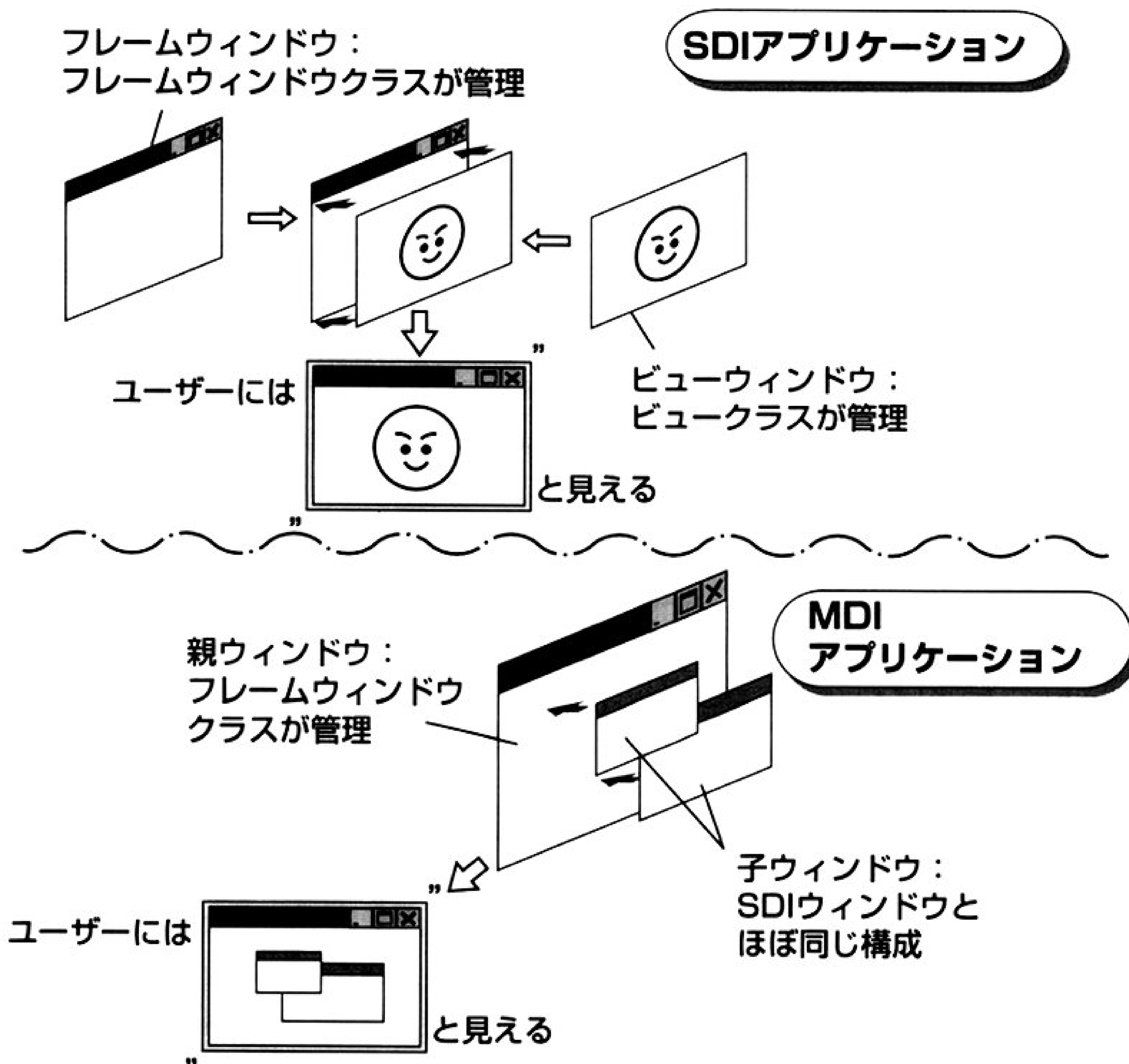


図 3-7 ビジュアルな視点から見たプログラムの構成

入力も管理します。つまり、ビュークラスはユーザーインターフェイスを管理するためのクラスだということができるでしょう。ただ、注意して欲しいのは実際にフレームウィンドウを管理するのはフレームウィンドウクラスのオブジェクトですし、アプリケーションのクライアント領域を管理するのはビュークラスのオブジェクトということです。また、ビュークラスのオブジェクトが管理するのはフレームウィンドウの上に重なって表示されているビューオブジェクト自身のクライアント領域であることも忘れないでください。本書では、以降クライアント領域とはビューオブジェクトのクライアント領域を指すものとします。

● ウィンドウは中身も肝心

次にプログラミングの視点から Windows アプリケーションを探ってみましょう。今度はさっきよりも 2 つクラスが増えています（**図 3-8**）。ということは、増えた 2 つのクラスは画面上に現れない部分を管理していることが想像できます。

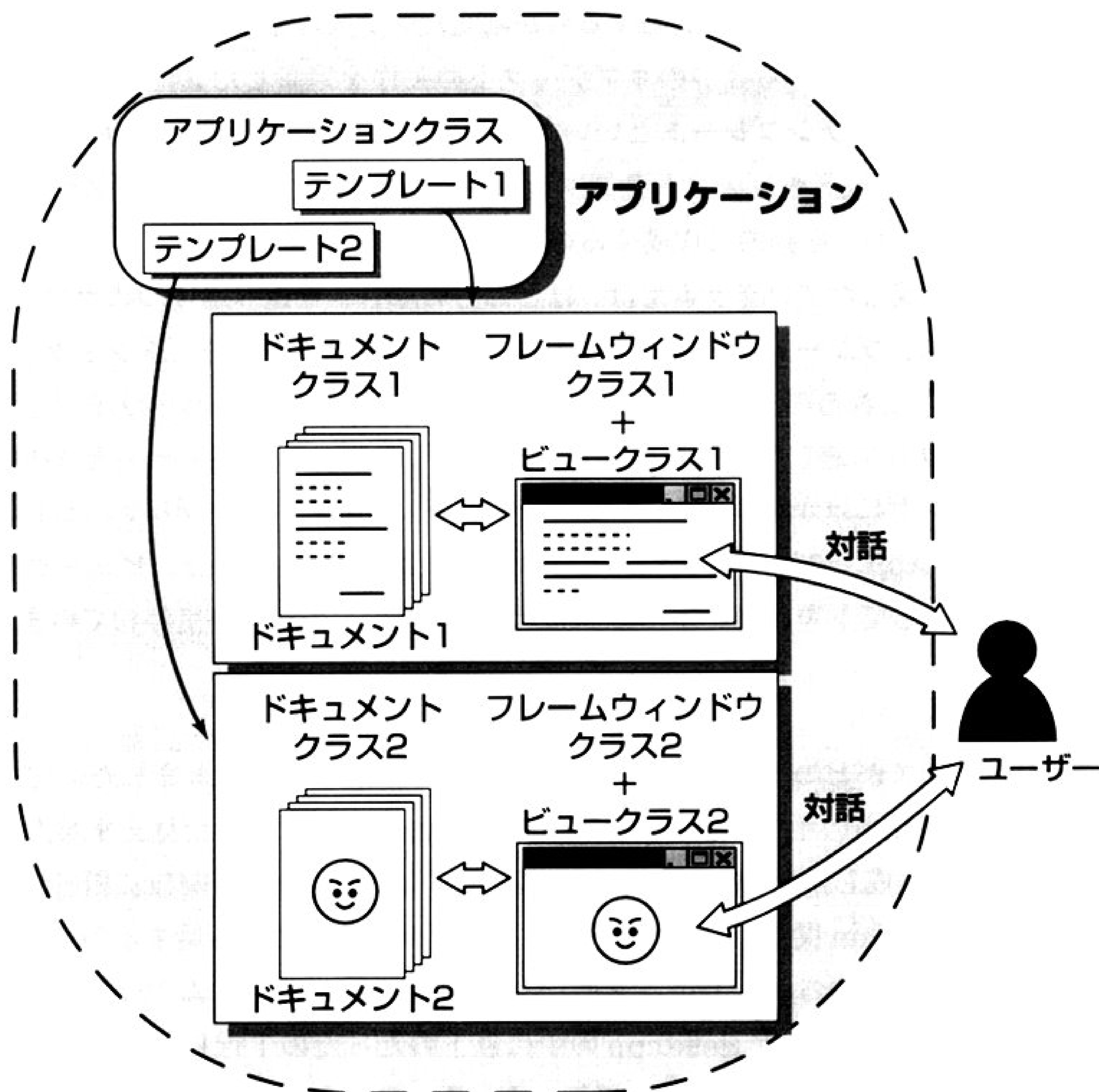


図3-8 プログラミングの視点から見たプログラムの構成

増えた2つのクラスのうち、**ドキュメントクラス**はその名前のとおりドキュメントを管理するためのクラスです。ただし、ここでいうドキュメントとは、アプリケーションが管理するデータのことです。たとえば、テキストエディタで扱うドキュメントはテキストデータですし、グラフィックスエディタで扱うドキュメントはグラフィックスデータです。

また、この視点から、アプリケーションの動作を考えてみると、ユーザーがアプリケーションを利用するということは、

1. ユーザーがビューとフレームからなるウィンドウに対して操作を行う
2. その操作によって、アプリケーションが管理しているドキュメントの内容が変更される
3. ドキュメントの変更をクライアント領域（すなわち、ビュー）を再描画することでユーザーに通知する

という作業の繰り返しだと考えることができます。この互いに関連するドキュメント／

ビュー／フレームウィンドウの組を管理するのが増えたクラスのもう1つ、**アプリケーションクラス**です。そして、これら3つのオブジェクトの動作を定義した3つのクラスをまとめたものを**ドキュメントテンプレート**といいます。アプリケーションクラスのオブジェクトは、必要に応じてこのドキュメントテンプレートを参照して、3つのクラスのオブジェクト（つまりウィンドウ）を動的に作成するのです。

AppWizardで作成したプログラムでは、ほとんどの場合、今述べた4つのクラス（アプリケーションクラス、フレームウィンドウクラス、ビュークラス、ドキュメントクラス）が最低限必要です*5。これらのクラスに機能を追加したり、関数をオーバーライドしたりして、自分のプログラムに適したクラスに作り替えていくことがVisual C++とMFCを利用したプログラミングにほかなりません。Helloプロジェクトでは、アプリケーションクラスとしてはCHelloApp、フレームウィンドウクラスにはCMainFrame、ビュークラスにはCHelloView、そしてドキュメントクラスにはCHelloDocクラスが使われています。

● オブジェクト

さて、4つのクラスがどのようにして絡み合っているのかは理解できました。でも、これらが実際にプログラムの中でいつどのように利用されるのかは、まだ見えません。しかも、AppWizardが生成したソースファイルにはどこを探してもmain関数に相当するもの（WindowsではWinMain関数）が見当たらず、プログラムがどこから始まるのか、まったくわかりません。それどころか、AppWizardが生成したコードのほとんどはクラスの定義ばかりでオブジェクトの定義はHello.cppの中に以下のたったの1行しかありません。これでどうしてプログラムが実行できるのでしょうか？

```
/////////////////////////////////////////
// 唯一の CHelloApp オブジェクト

CHelloApp theApp;
```

実はこのコードからすべては始まるのです。といってもよくはわかりませんが、要するにVisual C++でのプログラミングの世界では、アプリケーションの実行にはアプリケーションクラスのオブジェクトが1つあればいいのです。アプリケーションもオブジェクトだと思えば、これは当たり前のことのような気もします（しませんか?）。実際にアプリケーションがどのように実行されていくかについては、このあとの4章「フレームワークを解剖してみよう」で詳しく解説することにして、ここでは残る3つのクラスのオブジェクトがどのように定義されるのかについて話を続けることにしましょう。

---

*5　AppWizardを使わずにMFCを利用したプログラムを作成したり、AppWizardでダイアログベースのアプリケーションのスケルトンを作成した場合、およびAppWizardでドキュメント－ビュー・アーキテクチャのサポートのチェックをはずした場合は、この限りではない。

残りの3つのクラスはアプリケーションクラスの初期化時に、以下のようにAddDocTemplate関数を使ってドキュメントテンプレートリストに登録されます(Hello.cpp内で登録されている)。そのあとは、さきほどもいったように、必要に応じてアプリケーションクラスのオブジェクトが3つのクラスのオブジェクトを作成します。このことからわかるのは、ドキュメントテンプレートを構成する3つのクラスは切っても切れない仲であって、3つのクラスがそろって初めて一人前の仕事ができるということです。

```
CSingleDocTemplate* pDocTemplate;
pDocTemplate = new CSingleDocTemplate(
  IDR_MAINFRAME,
  RUNTIME_CLASS(CHelloDoc),
  RUNTIME_CLASS(CMainFrame),       // メイン SDI フレーム ウィンドウ
  RUNTIME_CLASS(CHelloView));
AddDocTemplate(pDocTemplate);
```

●ドキュメントービュー・アーキテクチャ

ここまでの説明を読み進めてきて、「なぜドキュメントとビューを別々に管理しなければならないのか?」ということを疑問に思った方も多いのではないでしょうか? 従来のプログラミングでは、データの管理とユーザーインターフェイスは特別分離して考えられるものではありませんでした。なぜMFCでは明確に分けて管理を行うのでしょう?

たとえば、ワードプロセッシングソフトの場合を考えてみます。ここで扱うドキュメントは書式指定の付いたテキストファイルです。文書(ドキュメント)の編集中はフォント指定や細かなレイアウト情報は無視してもいいから、単なるテキストファイルとして画面に表示できると編集の効率が上がりそうです。一方、編集が済んだあとのレイアウト時には、フォントの指定、図版の入り方など、できる限り出力機への出力と同等なものを画面に出力させたいはずです。ドキュメントの内容は同じでも、目的や必要に応じてその表現方法は複数存在します。このようなときには、やはりドキュメントとビューは別なものとして管理して、必要に応じてそのビュー(表現)を切り替える方法が有効のように思えます。

このように、ドキュメントとビューを別々に管理することには「アプリケーションが管理するデータを目的と必要に応じていろいろな形で表現できる」という利点があるのです。

MFCでは1つのドキュメントに対して、複数のビューを実装する機構が用意されています。これを**ドキュメントービュー・アーキテクチャ**といいます。ドキュメントービュー・アーキテクチャについては、第3部で実際にプログラムを作成しながら説明をします。

4 フレームワークを解剖してみよう

これまでの章ではVisual C++を使ったプログラムが、MFCで定義されたいくつかのクラスとそれらのクラスを使った特定の処理の枠組み（フレームワーク）の中で動作することを説明しました。フレームワークは、Windowsから送られてきたメッセージを受け取ると、適切なメッセージハンドラを呼び出すらしいということはわかっても、いったいそれがどのようなしくみで行われているのか気になるところです。そこで本章では、ついにフレームワーク内部へとせまってみることにします。

ここでは2章で作ったHelloプロジェクトをプログラム例として扱うことにします。このプログラムが実際に動作するときには、どういった順序で、どういった処理が実行されるのか、プログラマが記述した処理だけではなくフレームワーク内部の動作も含めて追ってみることにしましょう。

なお、本章の話題はかなり難しいことなので、今すぐわかる必要はありません。が、MFCを用いてプログラミングを行う以上は、MFCを利用したアプリケーションの動作を知っていて損はありませんから、何度か目を通してぜひ理解してください。

4.1 プログラムの実行はWinMainから？

MS-DOSではプログラムはmain関数から実行が始まったように、Windowsプログラムの実行はWinMain関数から始まります。したがって、実行の流れにそってコードを読み進めていくとすると、まずはWinMain関数からです。といきたいところですが、AppWizardが生成したコードの中には、どこをどう探してもWinMain関数は見つかりません。

Q1 いったいどこからプログラムの実行は始まるのですか？
A1 CHelloApp クラスのオブジェクトがまず定義されるところからプログラムは始まります。

Windowsアプリケーションでは、WinMain関数からその実行は始まります。が、その前に以下のようにグローバルオブジェクト(theApp)が定義されていることを思いだしてください。

```
////////////////////////////////////////
// 唯一の CHelloApp オブジェクト

CHelloApp theApp;
```

したがって、WinMain関数が実行される前にCHelloAppクラスのオブジェクトを定義するためにCHelloAppクラスのコンストラクタが実行されることになります。しかし、実際には、以下のようにCHelloAppクラスのコンストラクタは何もしていません。

```
CHelloApp::CHelloApp()
{
  // TODO: この位置に構築用コードを追加してください。
  // ここに InitInstance 中の重要な初期化処理をすべて記述してください。
}
```

CHelloAppクラスはCWinAppクラスの派生クラスですから、CHelloApp::CHelloApp関数の実行の前にはCWinApp::CWinApp関数(CWinAppクラスのコンストラクタ。MFC内で定義されている)が呼び出されます。実は、ここでさまざまな初期化が行われるのです。

CWinApp::CWinApp関数が実行され、初期化が終了すると、いよいよWinMain関数の実行が始まります。3章では、アプリケーションの実行にはアプリケーションクラスのオブジェクトが1つあればよいなどといいましたが、やはり、アプリケーションの実行にはWinMain関数が必要なのです。

Q2 しかし、WinMain関数も見当たりませんが？
A2 WinMain関数はフレームワーク(MFC)の中で定義されています。

theAppが定義されたあとに実行されるWinMain関数は、実はフレームワークの中で定義されています。つまり、MFCを使って作ったWindowsアプリケーションは、すべて同じWinMain関数を使っているのです。ただし正確に解説すると、実際にはWinMain関数は定義されておらず、_tWinMain関数が定義され、#defineによってこの関数がWinMain関数として扱われています。さらに、_tWinMain関数ではAfxWinMain関数を呼び出す、1行が記述されているだけなので、WinMain関数の実体はAfxWinMain関数ともいえます。こうした込み入った事情はありますが、ここではWinMain関数をMFCアプリケーションでのスタートアップ関数として解説を続けます。

フレームワークが提供するWinMain関数は、非常にシンプルな構成で、フレームワーク内部の初期化処理を除けば、実行する関数は、CWinAppクラスのメンバ関数、

- InitApplication関数
- InitInstance関数
- Run関数

の3つだけです。WinMain関数は、このように決まりきった関数しか呼び出さず、しかもそのコードはMFCの中で定義されていて、再定義することはできません。とすれば、作成したアプリケーションはこの中からどのようにして実行されるのでしょうか？ 実はこのあたりがC++言語の妙であり、真価を発揮するところなのです。詳細はAppendix Aに譲りますが、C++言語の仮想関数という機能を利用すれば、実際に呼び出される関数を後から差し替えることができるのです。すでにコンパイル済みの関数であってもそれは可能です。つまり、普遍的に使われるWinMain関数はすでにライブラリという形でコンパイルが済んでしまっているにもかかわらず、ユーザーが作成した関数を呼び出すことが可能ということです。

Q3 InitApplication関数とInitInstance関数を差し替えて、アプリケーションの初期化をするにはどうすればよいのですか？
A3 CWinAppクラスの派生クラスでInitInstance関数をオーバーライドします。

実は、CWinApp::InitApplication関数とCWinApp::InitInstance関数は仮想関数として定義されています。仮想関数とは、簡単にいってしまえば、プログラム実行時の実際のオブジェクトの型によって、実行される関数が決定する関数のことです（詳細はAppendix Aを参照）。したがって、CWinAppクラスの派生クラス（HelloプロジェクトではCHelloApp

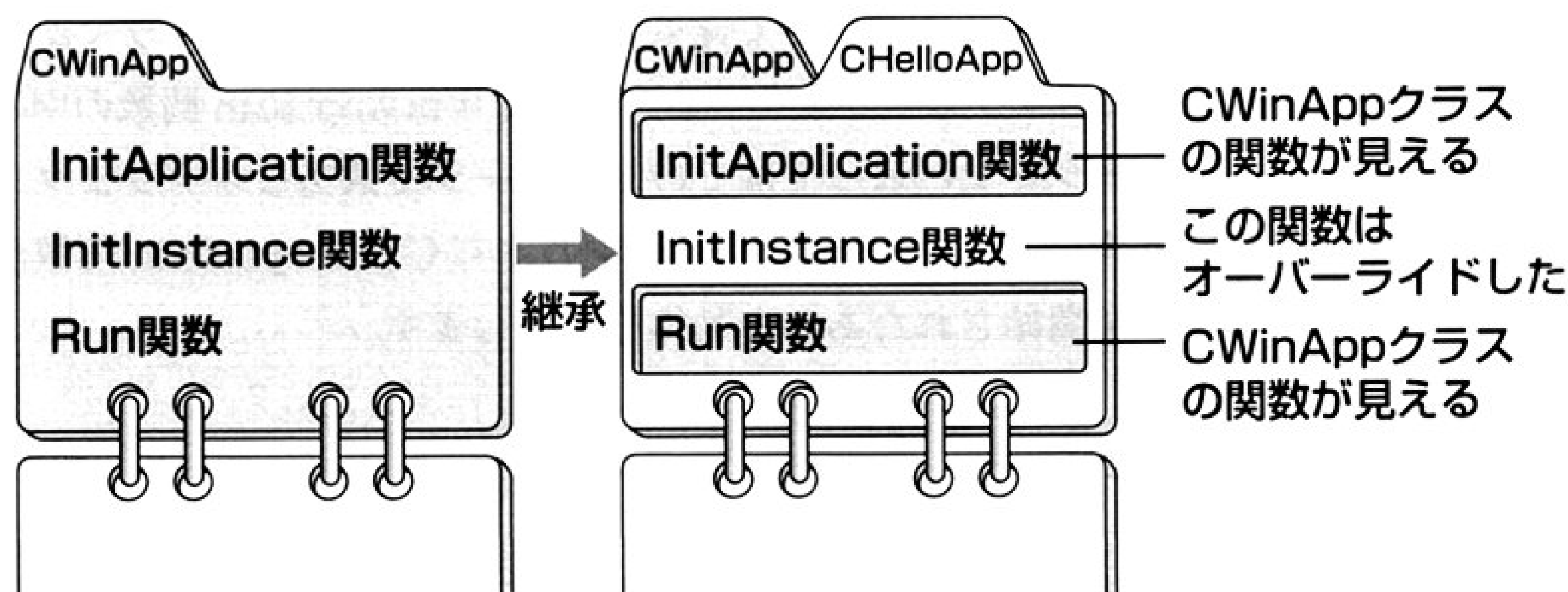


図4-1 仮想関数のオーバーライド

クラス）で InitApplication 関数と InitInstance 関数をオーバーライドして、そこに初期化処理を記述すればよいのです（**図 4-1**）。WinMain 関数は CWinApp クラスのオブジェクト（afxCurrentWinApp）に対して InitApplication 関数と InitInstance 関数を実行しようとしますが、オブジェクトの実際の型はプログラマが定義した派生クラスの型（CHelloApp クラス）ですから、ここでオーバーライドした InitApplication 関数と InitInstance 関数が呼び出され、うまい具合いに初期化が行われることになります。

こうして CWinApp クラスの 2 つのメンバ関数によってアプリケーションの初期化が行われますが、実際に利用するのは CWinApp::InitInstance 関数だけです。以前は。同じアプリケーションを複数同時に起動する場合に、最初の 1 つを起動したときにだけ必要になる初期化作業を行うために CWinApp::InitApplication 関数を使用しましたが、現在では使われていません。現在ではアプリケーションを起動するたびに、常に CWinApp::InitApplication 関数が呼び出されてしまうからです。この動作は CWinApp::InitInstance 関数と同じですから、今では不要のメンバ関数となっています。なお、CWinApp::InitApplication メンバ関数は現在でも残されてはいますが、これは互換性を保つためにすぎません。

Q4 InitApplication 関数や InitInstance 関数を実行したあとはどうなるのですか？
A4 Run 関数を呼び出し、メッセージループを実行し続けます。

Windows アプリケーションでは、Windows から届くメッセージを処理しない限り、まったく身動きすることさえできません。そこで、すべての Windows アプリケーションには、メッセージループと呼ばれるメッセージを処理するループが必要になります。メッセージループの目的は、Windows からメッセージを受け取り、そのメッセージを処理できるオブジェクトに送り付けることです。オブジェクトは、受け取ったメッセージを判断して適切な関数（メッセージハンドラ）を実行します。プログラマが記述したコードのほとんどはこのメッセージループの中から呼び出されるのです。

メッセージループはフレームワークによって提供されているので、プログラマがわざわざ記述する必要はありません。メッセージループの実体は CWinApp::Run 関数の中にある無限ループであり、メッセージを受け取ってはそのメッセージを適当なオブジェクトに送り付ける処理を繰り返します。次節では、WinMain 関数から CWinApp::Run 関数が呼び出され、メッセージループが開始されたあとの動作を説明します。

4.2 メッセージの処理

前節では、アプリケーションが起動してから、メッセージループにたどり着くまでの過程を説明しました。しかし、メッセージループの前に行う処理はあくまでも初期設定で、Windows アプリケーションの中心はメッセージループの間に呼び出されるメッセージハンドラにあることはすでにおわかりでしょう。本節では、Hello プロジェクトで記述した CHelloView::OnDraw 関数を例に、メッセージを受け取ってからメッセージハンドラが呼び出されるまでの流れを説明します。

純粋仮想関数

CView クラスの定義では、CView::OnDraw 関数のプロトタイプ宣言は以下のようになっています。

```
virtual void OnDraw(CDC* pDC) = 0;
```

プロトタイプ宣言の後ろの「= 0」は、このメンバ関数が純粋仮想関数であることを示しています。純粋仮想関数を含んでいるクラスのオブジェクトは作ることはできません。かならずそのクラスの派生クラスを定義して純粋仮想関数をオーバーライドしなければなりません。

Q1 AppWizard が生成したコードでは、どこからも CHelloView::OnDraw 関数が呼び出されていないのに、なぜこの関数が実行されるのですか？
A1 フレームワークの中に OnDraw 関数を呼び出すコードがあるからです。

CView クラスのメンバ関数の 1 つに OnPaint 関数があります。CView::OnPaint 関数はデフォルトでは、OnDraw 関数を呼び出します。そして、このとき実際に呼び出されるのがオーバーライドをした CHelloView::OnDraw 関数なのです。

Q2 いつ CView::OnPaint 関数が呼び出されるのですか？
A2 画面の描画を要求するメッセージ（WM_PAINT）が Windows からアプリケーションに送られてきたときです。

メッセージが Windows から送られてきたときに呼び出される関数をメッセージハンドラと呼ぶことはすでに説明しました。Windows には非常に数多くのメッセージが定義されていますが、その 1 つ 1 つにメッセージハンドラが 1 対 1 に対応しています。CView::OnPaint 関数もメッセージハンドラの 1 つで、WM_PAINT メッセージに対応しています。

つまり、Windows から WM_PAINT メッセージがアプリケーションに送られてくると、フレームワークによって CView::OnPaint 関数が呼び出されるというわけです。

Q3 WM_PAINT メッセージはどのようなときに送られてくるのでしょう?
A3 ウィンドウの再描画が必要になったときです。

WM_PAINT メッセージは、ウィンドウの再描画が必要になったことを知らせるために、Windows から送られます。再描画が必要になるときとは、ウィンドウを開くときや、他のウィンドウの下になって隠されていた部分が見えるようになったときや、アプリケーションの中で Windows に描画を要求したときなどです。

Q4 メッセージとメッセージハンドラの関係がよくわかりません。
A4 メッセージに対応するメッセージハンドラはメッセージマップで定義されています。

すべてのメッセージにはデフォルトのメッセージハンドラがフレームワークによって用意されていますが、メッセージハンドラはプログラマが自由に変更(オーバーライド)することができます。しかし、メッセージを受け取って、対応するメッセージハンドラを呼び出すのはフレームワークの仕事ですから、プログラマはメッセージハンドラの変更をフレームワークに伝えなければいけません。プログラマがメッセージハンドラの変更をフレームワークに伝えるために定義するテーブルを**メッセージマップ**と呼びます。

ClassWizard を使ってメッセージハンドラを作成する場合は、プログラマがメッセージマップを直接編集する必要はほとんどありません。しかし、残念ながら ClassWizard がすべてのメッセージを管理できるわけではありませんし、また誤ってメッセージマップを壊してしまったときなどのためにも、メッセージマップの構造を知っておいた方がいいでしょう。

ここでは CMainFrame クラスのメッセージマップを例にとって説明します。以下は、CMDIFrameWnd クラスを基底クラスに持つ CMainFrame クラスのメッセージマップです。このように、メッセージマップはクラスごとに定義します。

```
BEGIN_MESSAGE_MAP(CMainFrame, CMDIFrameWnd)
  //{{AFX_MSG_MAP(CMainFrame)
  ON_WM_CREATE()                // WM_CREATE メッセージを処理する
  //}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

CMainFrame クラスが独自に処理をするのは、WM_CREATE メッセージだけです。これを表しているのが、ON_WM_CREATE マクロです。このように、メッセージ名の前に「ON_」を追加した名前のマクロを使って、処理するメッセージを指定します。また、

「WM_」で始まるメッセージに対応するメッセージハンドラの名前は、あらかじめ決まっていて変更することはできません。たとえば、WM_CREATEメッセージならば、OnCreate関数がメッセージハンドラとなります。このように、メッセージ名から「WM_」を取り除き、代わりに「On」を付け加えたものがメッセージハンドラの名前となっています。

メッセージマップに新しいエントリを追加したら、忘れずにメンバ関数のプロトタイプ宣言もしておかなければいけません。フレームワークにとってはメッセージハンドラという特別な関数であっても、コンパイラにとっては通常のメンバ関数と変わらないのです。

```
class CMainFrame : public CMDIFrameWnd
{
...略
// 生成したメッセージマップ関数
protected:
  //{{AFX_MSG(CMainFrame)
  afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
  //}}AFX_MSG
  DECLARE_MESSAGE_MAP()
};
```

WM_CREATEメッセージに対応するメンバ関数のプロトタイプ宣言は、「afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);」となります。メッセージハンドラの引数と返り値は**表4-1**にあげるように、メッセージハンドラごとにさまざまですから、覚えきることは事実上不可能ともいえます。その点、ClassWizardを使えばメッセージを選ぶだけで、メッセージマップ／メンバ関数としてのプロトタイプ宣言／メッセージハンドラの定義をすべて自動的に行ってくれます。

メッセージ	メッセージマップ	メッセージハンドラ
WM_PAINT	ON_WM_PAINT()	void OnPaint()
WM_LBUTTONDOWN	ON_WM_LBUTTONDOWN()	void OnLButtonDown(UINT nFlags, CPoint point)
WM_CHAR	ON_WM_CHAR()	void OnChar(UINT nChar, UINT nRepCnt, UINT nFlags)

表4-1 メッセージとメッセージハンドラの対応例

Q5 メッセージハンドラが終了したあとは、どうなるのですか？
A5 いったんWindowsに処理を戻し、またメッセージループを繰り返します。

メッセージハンドラを作るときにかならず考慮しなければならない重要な指針が1つあります。それは、「できるだけ短い時間で処理を終了させる」ということです。仮にいつまでも1つのメッセージハンドラの中で処理を続けていると、次にやってきたメッセージは

処理されず、溜まっていく一方になってしまいます。つまり、ウィンドウを動かそうとしたり、キー入力をしようとしてもまったく反応できず、ユーザーの目にはあたかもハングアップしたかのように見えてしまうのです。このことはプログラミングを行う際には常に気に留めておくようにしてください。

メッセージハンドラが終了すれば、フレームワークが自動的に処理を Windows に戻してくれます。そして、アプリケーションはまた次のメッセージが届くのをじっと待ち続けるのです。

Q6 いつメッセージループは終了するのですか？
A6 WM_QUIT メッセージを受け取ったときです。

ウィンドウ右上の＜閉じる＞ボタンをクリックしたり、タイトルバーの左にあるアイコンをダブルクリックしたり、あるいはメニューから［ファイル］－［終了］を選択したりすると、さまざまな経路をたどりますが、最終的には WM_QUIT メッセージがアプリケーションに送られます。WM_QUIT メッセージはメッセージループで特別な扱いをされていて、メッセージハンドラは呼び出されず、代わりにメッセージループを終了します。続いて、CWinApp::Run 関数、WinMain 関数と順に抜け出して、最終的にアプリケーションが終了します。

4.3 プログラムの流れ

長い道のりでしたが、やっとスケルトンに追加したたった1行のコードにたどり着くまでの流れをつかむことができましたね。ここでもう一度、Hello 全体の流れを**図 4-2** にまとめてみてみましょう。図の左側がフレームワーク内の処理、右側が AppWizard が生成したコードとプログラマが記述したコードです。

図 4-2 に示す Hello の実行の手順の中でとくに注意して欲しいのは以下のようなことです。

- **アプリケーションクラスのオブジェクトの定義部**
  CHelloApp クラスのオブジェクトを定義する。プログラマが記述したコード中でまったく利用していなくても、フレームワークが参照するので、かならず1つ定義しておくこと。
- **CWinApp::CWinApp 関数**
  CHelloApp クラスのコンストラクタが実行される前に、その基底クラスのコンストラクタが実行される。この中では、主に CWinApp クラスのメンバ変数やグローバル変

数の初期化が行われる。

● **WinMain 関数**

グローバルオブジェクトのコンストラクタの次に呼び出されるスタートアップ関数。InitApplication 関数、InitInstance 関数、Run 関数を順に呼び出す。

● **CHelloApp::InitInstance 関数**

インスタンスごとに必要な初期化処理（ドキュメントテンプレートの登録、フレームウィンドウの作成、コマンドラインの解析など）が行われる。

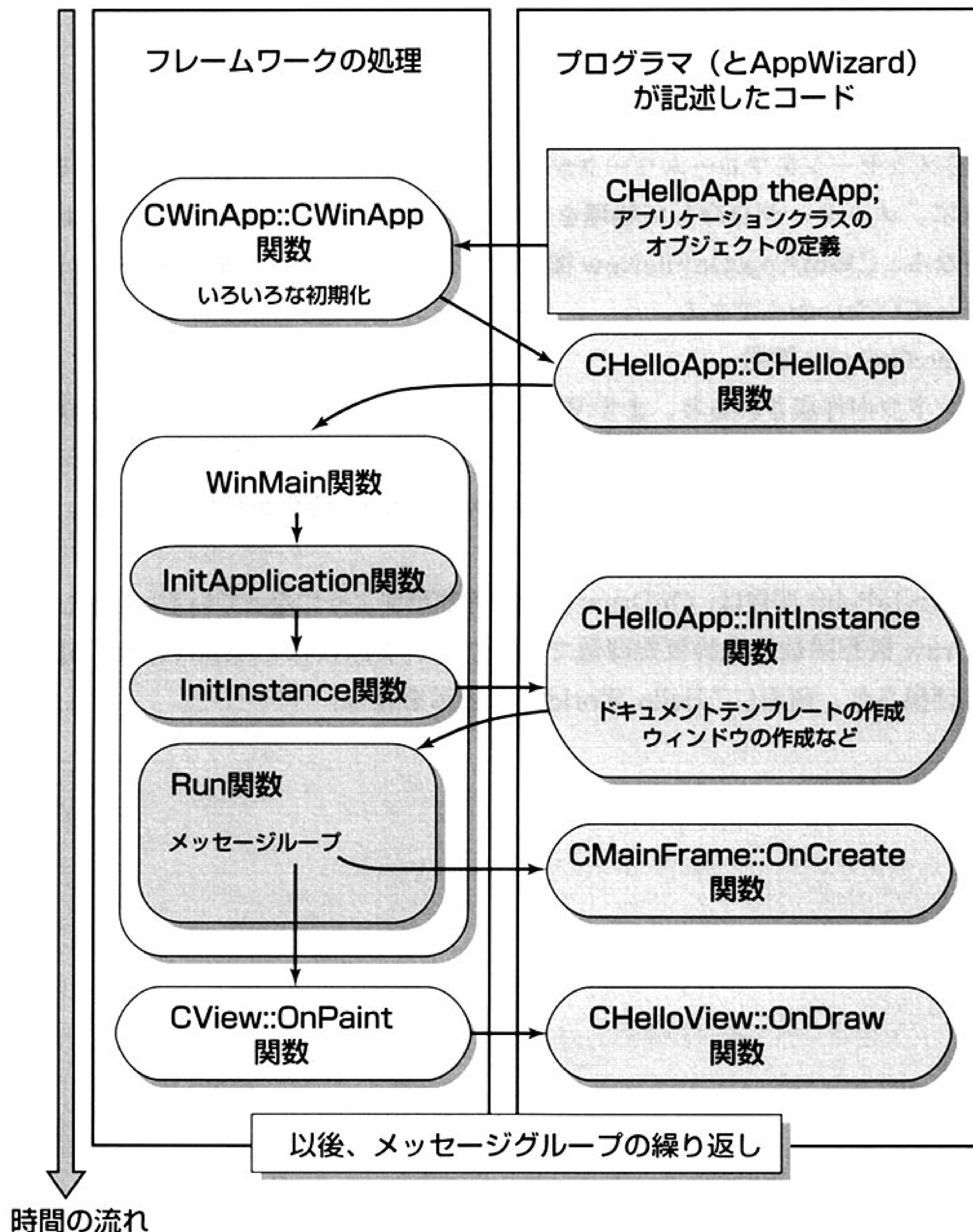


図 4-2 Hello の流れ

● **CWinApp::OnFileNew 関数**

図 4-2 には示していないが、CHelloApp::InitInstance 関数から、SDI アプリケーションのメインフレームウィンドウを作成するために呼び出される。MDI アプリケーションの場合は、この関数を呼び出す前に、CFrameWnd::LoadFrame 関数で親ウィンドウを作成しておく必要がある。

● **CWinApp::Run 関数**

WinMain 関数から呼び出され、メッセージループを実行する。以後アプリケーションが終了するまで、Windows メッセージを受信、メッセージハンドラの呼び出しを繰り返す。

● **CMainFrame::OnCreate 関数**

CWinApp::OnFileNew 関数でウィンドウを作成したときに発行された WM_CREATE メッセージをフレームワークが受け取り、このメンバ関数が呼び出される。このように、メッセージを介して処理を行う場合には、すぐに処理されるとは限らない。なぜなら、CWinApp::OnFileNew 関数を実行した時点では、まだメッセージループが始まっていないからである。

● **CView::OnPaint 関数**

ウィンドウが作成されると、まず WM_PAINT メッセージが Windows から送られてくる。このメッセージを受け取ったフレームワークは、対応するメッセージハンドラ、OnPaint 関数を呼び出す。実際には CView::OnPaint 関数が呼び出される。

● **CHelloView::OnDraw 関数**

CView::OnPaint 関数は、OnDraw 関数を呼び出すようになっている。しかし、CView::OnDraw 仮想関数は純粋仮想関数であるため、実際には CHelloView::OnDraw 関数が呼び出され、画面に "Hello World!" と表示を行う。

第2部
Visual C＋＋プログラミングの基本を押さえよう

第１部では、Visual C＋＋のプログラムがどのように組み立てられていくのか、その基本ともいうべき部分を紹介しました。こうして第一歩を踏み出したあなたの前には、広大なWindowsプログラミングの世界が待ち受けています。しかし、Visual C＋＋を使って本格的な仕事を始めるには、まだまだ準備が足りません。とくに今のあなたに欠けていると思われるのは、Windows特有のグラフィカルな環境でデータ入出力を行うための知識です。そこでこの第２部では、GDIを使ったデータ出力と、メニューとダイアログボックスを利用したデータ入力、さらにキーボードとマウスからのデータ入力を中心にWindowsのユーザーインターフェイスについて解説します。さらにリソースエディタとClassWizardの詳しい使い方も覚えてもらいます。そして、最後にプログラムを作成する上で欠かせないこと――すなわち、デバッグ――についても解説をすることにしましょう。ここで述べられる情報は、いわばWindowsプログラミングをマスターするための基礎体力に相当します。しっかり身に付ければ、Visual C＋＋を征服するための大きな力となってくれることでしょう。

1 GDIはグラフィックス表示の合言葉

MS-DOSとWindowsの画面を比較したとき、両者のもっとも大きな相違点は何でしょう？ 考えるまでもなく、答は解像度の違いです。かたや80文字×25行程度のテキスト画面、かたや最低でも640ドット×480ドットの広大なグラフィックス画面が相手です。画面に表示される絵や文字の細かさ、美しさは、どうしてもWindowsに軍配が上がります。

もちろんMS-DOSのアプリケーションでも独自にグラフィックスを扱うものはありますが、WindowsはOS自体がグラフィックス処理機能を装備していることが大きな強みです。プログラマはWindowsの持つ機能を利用できるため、面倒な作業はほとんど必要なく、わずかな労力で、Windows特有のグラフィカルなユーザーインターフェイスが実現できるのです。

Windowsが用意した、グラフィックスデータを扱うための統一インターフェイスには、**GDI（Graphics Device Interface）**という名前が付けられています。GDIは、その名のとおり画面だけではなく、プリンタ出力などを含む広範囲のグラフィックスデバイス操作ルーチンの集合体ですが、この章ではディスプレイ画面の扱いに話を限定して、GDIの基本操作を学ぶことにします。

1.1 デバイスコンテキストとCDCクラス

GDIという統一インターフェイスの中心的存在が**デバイスコンテキスト**です。デバイスコンテキストは、しばしばDCとも略されます。

デバイスコンテキストは、いろいろな種類のグラフィックス表示デバイスを抽象化したものです。抽象化という言葉がわかりにくければ、ハードウェアの差異を吸収するクッションとでもいいましょうか。

たとえばMS-DOSでは、CONファイルにデータを書き込むことで、データの画面出力が可能なことをご存じかと思います。これができるのは、画面とディスクのように、ハードウェアは異なっても、「ファイル」という名前のもとに両者を統一的に扱えるように、MS-DOSの中でソフトウェア的なクッションが設けられているからです。

デバイスコンテキストにも、これと同じような意味合いがあります。Windowsでは、グラフィックスデータの出力先は画面やプリンタなどの具体的なハードウェアではなく、常にデバイスコンテキストです。実際にはそれは画面上のウィンドウを意味するかもしれないし、プリンタかも、あるいはメモリ内のバッファ領域かもしれません。しかしデータを出力する側は、そんなことは気にせず、いつでもデバイスコンテキストに対してデータを出力すればよいのです。つまり、デバイスコンテキストを利用することによって、Windowsでは、デバイスに依存せずにグラフィックスを出力できるようになるのです。

デバイスコンテキストの取得と解放

デバイスコンテキストの実体は、画面のサイズや描画モードの設定など、実際のデバイスの動作を決めるデータの集まりです。Visual C++では、CDCクラス(デバイスコンテキストクラス)を使ってデバイスコンテキストを表します。

Windowsのプログラムで何らかのデータ出力を行うには、まず出力対象のデバイスコンテキストを取得し、次にそのデバイスコンテキストに対してCDCクラスのメンバ関数を実行するという手順を踏みます(コラム参照)。たとえばクライアント領域に文字や図形を表示する場合も、とにかく最初にそのデバイスコンテキストを得なければなりません。そこで利用されるのが、CWnd::GetDC関数と、CWnd::ReleaseDC関数の2つです。

CWnd::GetDC関数は、指定したウィンドウのクライアント領域のデバイスコンテキストを取得します。また、この関数で得たデバイスコンテキストは使い終えた時点で解放する必要がありますが、それを行うのがCWnd::ReleaseDC関数です。

以下に示すプログラムは、CMyViewというビュークラスがあるものとして、そのクライアント領域に"Hello World!"と表示する、CMyView::ShowHello関数です。

```
void CMyView::ShowHello()
{
  CDC*  pDC;

  pDC = GetDC();          // pDC = this->GetDC(); と同等(コラム参照)
  pDC->TextOut(0, 0, "Hello World!");
  ReleaseDC(pDC);         // this->ReleaseDC(); と同等( 〃 )
}
```

メンバ関数の呼び出し

メンバ関数は、特定のクラスのオブジェクトを操作するために定義された関数で、C言語には存在しなかった概念です。たとえば本文中に登場したGetDC関数とReleaseDC関数がそうです。メンバ関数を実行する場合は、原則として、操作の対象となるオブジェクトか、オブジェクトへのポインタを指定しなければいけません。オブジェクトそのものを対象とする場合は、クラスオブジェクト選択子「.」を、オブジェクトへのポインタを指定する場合は、クラスポインタ選択子「->」を使います。

```
// オブジェクトを指定する
CDC*  pDC;
CWnd  Wnd;
pDC = Wnd.GetDC();   // Wndを対象としてGetDC関数を実行する

// オブジェクトへのポインタを指定する
CDC*  pDC;
CWnd* pWnd;
pDC = pWnd->GetDC();   // pWndの指すオブジェクトを対象にGetDC関数を実行
```

メンバ関数の中では「そのメンバ関数の操作対象となっているオブジェクト」をthisというポインタが指しています。よってCWndクラス(またはCWndクラスの派生クラス)のメンバ関数の中では、次のようにthisを介してGetDC関数を実行できます。この場合「this->」は省略できます。

```
void CWnd::Foo()         // CWndクラスのメンバ関数Fooの定義
{
  CDC*  pDC;
  pDC = this->GetDC();   // Foo関数の対象のオブジェクトのGetDC関数を実行
  pDC = GetDC();         // pDC = this->GetDC(); と同等である
  ...
}
```

さらに詳しいことについては、Appendix Aを参照してください。

● OnDraw関数の引数として渡されるデバイスコンテキスト

デバイスコンテキストは、フレームワークの中で自動的に取得される場合もあります。たとえば第1部のHelloプロジェクトで作成したOnDraw関数の定義を見てみましょう。

```
void CHelloView::OnDraw(CDC* pDC)
{
  pDC->TextOut(0, 0, "Hello World!");
}
```

このOnDraw関数では、わざわざ自分でデバイスコンテキストを取得する必要はありません。引数のpDCに対してデータを出力すれば、その結果として画面に文字が表示されるのです。これは、フレームワークの中でデバイスコンテキストが取得され、それがOnDraw関数に引数pDCとして渡されているからです。

デバイスコンテキストの取得に関してOnDraw関数がこのような特別な立場をとっていることには理由があります。OnDraw関数は実は画面出力だけではなく、プリンタ出力の際にも利用される関数だからです。つまりOnDraw関数がデバイスコンテキストを引数としているのは、画面出力時には画面のデバイスコンテキスト、プリンタ出力の場合はプリンタのデバイスコンテキストが指定できるようにするためなのです。

まあ理由はともあれ、今のところは「OnDraw関数の中ではGetDC関数やReleaseDC関数を実行する必要はない」ということを覚えておいてください。

● 文字列表示の基本操作

ここでデバイスコンテキストに文字列を表示する基本操作、つまりCDCクラスのメンバ関数をまとめておきましょう。最初にあげられるのは、すでにおなじみのCDC::TextOut関数です。この関数は、文字を表示する位置、表示したい文字列、そして文字列の長さを引数とします。

```
char *str1 = "Hello No.1";
pDC->TextOut(x, y, str1, strlen(str1)); // 座標(x,y)に"Hello No.1"を表示
```

引数にCStringクラスのオブジェクトを指定することもできます。この場合は文字列の長さを指定する必要はありません。CStringクラスとは、文字列を操作するためにMFCが用意したクラスでBASIC言語的な文字列操作(Mid関数／Left関数／Right関数など)が可能な便利なクラスです。

```
CString str2("Hello No.2");
pDC->TextOut(x, y, str2);     // 座標(x,y)に"Hello No.2"を表示
```

また文字列定数を指定する場合も、やはり長さの指定は必要ありません。

```
pDC->TextOut(x, y, "Hello No.3");    // 座標(x,y)に"Hello No.3"を表示
```

文字の色を変えるにはCDC::SetTextColor関数、文字のすきまから見える背景の表示方法を変えるにはCDC::SetBkMode関数とCDC::SetBkColor関数を利用します。初期設定は、文字色が黒、背景は不透明、背景色は白です。

```
pDC->SetTextColor(RGB(0, 0, 255));  // 文字の色を青に設定する
pDC->SetBkMode(TRANSPARENT);        // 文字の背景を透明にする
pDC->SetBkMode(OPAQUE);             // 文字の背景を不透明にする
pDC->SetBkColor(RGB(255, 0, 0));    // 文字の背景色を赤に設定する
```

画面に数値などを表示するには、いったん文字列化してからCDC::TextOut関数で出力します。数値の文字列化にはWindows APIとして提供されているwsprintf関数を利用するとよいでしょう。この関数はC言語の標準ライブラリのsprintf関数と同じ機能を持ちますが、Windowsカーネル内にDLLとして用意されているため、いろいろな点で有利になっています。wsprintf関数の利用例を次に示します。

```
char buf[20];
int x = 2;
int y = 3;

wsprintf(buf, "(%d,%d)", x, y);          // "(2,3)"という文字列を作成
pDC->TextOut(x, y, buf, strlen(buf));    // 座標(2,3)に"(2,3)"と表示
```

グラフィックス表示の基本操作

グラフィックスの表示方法も基本的には文字列表示とまったく同じで、デバイスコンテキストを対象として、グラフィック描画用メンバ関数を実行するだけです。たとえば点を表示するには次に示すCDC::SetPixel関数を使用します。

```
pDC->SetPixel(x, y, RGB(255, 0, 0)); // (x,y)に赤の点を打つ
```

SetPixel関数は、指定した座標に、指定した色で、点を表示します。このときデフォルトの設定では、ビューウィンドウの左上隅を(0,0)とするピクセル単位の座標系が使われます*1。

CDC::SetPixel関数の第3引数は、描画色を指定するCOLORREF型の数値です。Windowsで色の指定をする場合は、かならずこのデータ型を指定します。COLORREF型の数値を得るには、RGBマクロを利用して赤、緑、青の色成分を指定するか、またはWindows APIのGetSysColor関数によってWindowsのシステムカラーを得るのが簡単です。RGBマクロの具体的な利用方法はこのあとで説明します。

直線を引くには、CDC::MoveTo関数とCDC::LineTo関数を組み合わせて使用します。

```
pDC->MoveTo(x, y);  // (x,y)を線の始点とする
pDC->LineTo(x, y);  // (x,y)まで線を引く
```

まずMoveTo関数によって始点の位置を決め、次にLineTo関数で終点を指定すると、2点間に線が引かれます(ただし終点は直線内に含まれません)。LineTo関数を続けて実行し、連続した折れ線を描いていくこともできます。

ところで、CDC::LineTo関数には、CDC::SetPixel関数と違って直線の色を指定する引数がありません。一般にWindowsプログラムで線の色を変えるには、「ペン」というGDI

---

*1 CDC::SetMapMode関数を利用すれば他の座標系も指定可能だが、本書では使用しない。

オブジェクトを使用するからです。GDI オブジェクトとは何か? その答は、このあとすぐに……

1.2 GDIオブジェクト ——ペンとブラシとビットマップ

グラフィック処理は、どんなに複雑に見えても、結局はいくつかの基本操作を組み合わせたものにすぎません。そこで Windows では、デバイスコンテキストに対して行う操作を何種類かに大別し、それぞれを処理するための専用の「道具」を用意しました。この「道具」を総称して **GDI オブジェクト**と呼びます。

この項では、ペン、ブラシ、ビットマップという 3 種類の GDI オブジェクトの使い方を見ていきましょう。これらは**表 1-1** のような操作に対応しています。

GDI オブジェクト	操作
ペン	線を引く
ブラシ	面を塗る
ビットマップ	図形を表示する

表 1-1 GDI オブジェクトの種類

GDI オブジェクトは**図 1-1** に示す MFC のクラスを介して操作されます。たとえば、ペン、ブラシ、ビットマップに対しては、それぞれ CPen、CBrush、CBitmap というクラスが用意されています。また本書では扱いませんが、このほかに CFont(フォント)、CPalette

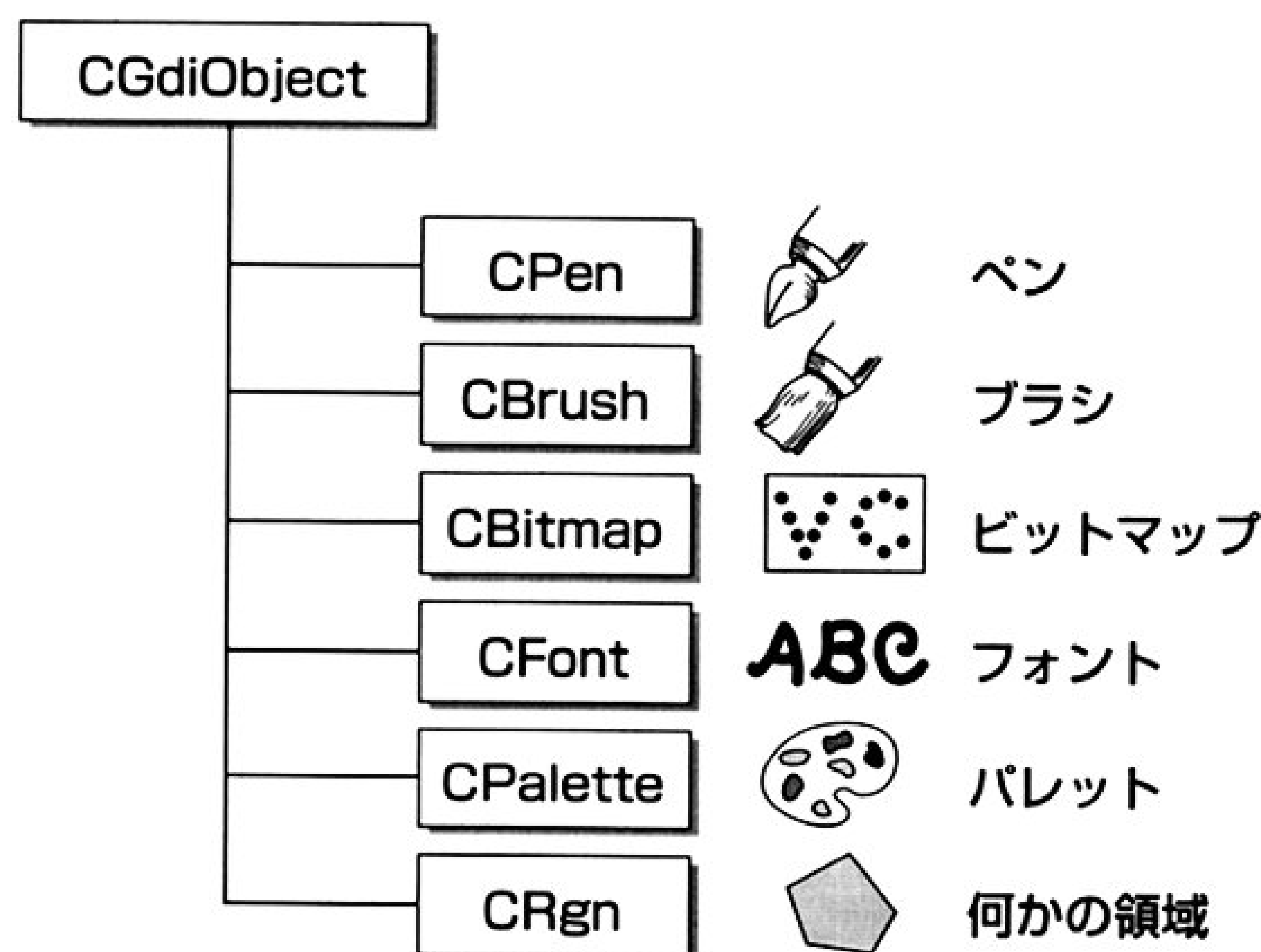


図 1-1 GDI オブジェクトを管理するクラス

(パレット)、CRgn(リージョン)という3つのクラスがあります。これらのクラスはすべて、CGdiObject クラスの派生クラスとして定義されています。

ここで、GDI オブジェクトは、C++言語でいうところの「クラスのオブジェクト」とは別ものだということに注意してください。

たとえばCPen クラスのオブジェクトといった場合、これは「ペンの GDI オブジェクト」ではなく、あくまでも「CPen というクラスのオブジェクト」です。実際の GDI オブジェクトは、CPen クラスのオブジェクトが管理しているデータの1つにすぎないのです。

そもそも C++言語と Windows が同じオブジェクトという用語を使ってしまったのが混乱のもとですが、いまさら変更することもできません。以下の項では「CPen クラスのオブジェクト」、「ペンの GDI オブジェクト」という具合にいい方を変え、できるだけ区別がつくよう努力してみました。

● ペンの利用法 ―― 線の描画スタイルを指定する

ペンは線の色や形状を指定します。非常に単純な GDI オブジェクトですが、以下に述べるペンの使い方の中には、GDI オブジェクト利用のためのエッセンスがすべて含まれています。まずはここで GDI オブジェクトの操作の基本を覚えましょう。

G1 プロジェクトの設計

それでは、ペンの GDI オブジェクトの実際を見て行く前に、テストプログラムを記述するためのプロジェクトを作成しておきましょう。以下は AppWizard と、そのオプションで定めるアプリケーションの枠組みです。

- プロジェクト名:G1
- アプリケーションのタイプ:SDI
- データベースのサポート:しない
- 複合ドキュメントのサポート:しない
- ツールバー/ステータスバー:なし
- 印刷と印刷プレビュー:なし
- そのほか:デフォルトのまま

上の方針に従って、AppWizard でプロジェクトを新規に開始してください。また、このプロジェクトは画面描画のテストのためのものですから、ここで必要になる関数は第1部でもお世話になったビュークラスの OnDraw 関数です。このプロジェクトでは、計5つのテストをしますから、PenTest1~PenTest5 までの5つの関数を OnDraw 関数の中から呼び出すことになります。そこで、G1View.cpp の**リスト 1-1** に示した部分を**リスト 1-2** のように修正しておいてください。これからは、PenTest1~PenTest5 までの関数の実装をしながら、ペンの利用方法を学んでいくことにしましょう。

リスト 1-1　変更前のリスト (G1View.cpp)

```
/////////////////////////////////////////
//// CG1Viewクラスの描画

void CG1View::OnDraw(CDC* pDC)
{
  CG1Doc* pDoc = GetDocument();

  // TODO: この場所にネイティブ データ用の描画コードを追加します。
}
```

リスト 1-2　変更後のリスト (G1View.cpp)

```
/////////////////////////////////////////
//// CG1Viewクラスの描画
void PenTest1(CDC* pDC)
{
}

void PenTest2(CDC* pDC)
{
}

void PenTest3(CDC* pDC)
{
}

void PenTest4(CDC* pDC)
{
}

void PenTest5(CDC* pDC)
{
}

void CG1View::OnDraw(CDC* pDC)
{
  pDC->TextOut(16,  10, "TEST 1");  PenTest1(pDC);
  pDC->TextOut(16,  30, "TEST 2");  PenTest2(pDC);
  pDC->TextOut(16,  60, "TEST 3");  PenTest3(pDC);
  pDC->TextOut(16, 170, "TEST 4");  PenTest4(pDC);
  pDC->TextOut(16, 195, "TEST 5");  PenTest5(pDC);
}
```

ペンの作成

ペンを利用する場合には、最初に CPen クラスのオブジェクトを定義します。CPen はその名前が示すとおり、ペンを扱うためのクラスです。

```
CPen  myPen;
```

この定義の段階では、まだ myPen に対応するペンの GDI オブジェクトは存在していません。ペンの GDI オブジェクトを作成するには、CPen::CreatePen 関数を実行します。

```
myPen.CreatePen(PS_SOLID, 1, RGB(255, 0, 0));
```

この結果、Windows のシステムメモリ(GDI リソース領域)内にペンが作成され、myPen と結び付けられます。CPen::CreatePen 関数の引数の意味は次項で説明しますが、とりあえず上の例では、実線、太さ 1、赤色のペンを指定しているものと理解してください。

さて、個々のデバイスコンテキストには、常に 1 本のペンが割り当てられています。初期設定は太さ 1 の黒い実線です。これを myPen に交換すると、以後そのデバイスコンテキストに引く線は、すべて myPen を用いて描かれることになります。

デバイスコンテキストのペンを交換するには、次のように CDC::SelectObject 関数を使用します。ここで pDC は目的のデバイスコンテキストへのポインタです。

```
CPen* pOldPen;
pOldPen = pDC->SelectObject(&myPen);
```

このとき、CDC::SelectObject 関数は、それまでデバイスコンテキストに割り当てられていたペンへのポインタを返します。ペンを使い終わったらもとの状態に戻す必要があるので、戻り値は上記のようにかならず記録しておいてください。

これで準備は完了です。以後デバイスコンテキスト pDC に対して引く線は、myPen で指定した「太さ 1 の赤い実線」になります。あとは、三角関数のグラフでも、エッチな絵でも、お望みのままに描けます。

描画を終えて用済みになったペンは削除しなければなりません。この作業を忘れてペンの作成を繰り返すと、Windows の GDI リソース領域が次第に食いつぶされていってしまいます。ペン削除の手順は以下のとおりです。

まず myPen をデバイスコンテキストから切り離すため、再び CDC::SelectObject 関数を実行して、さきほど記録しておいた pOldPen と myPen を交換します。この操作の目的は、あくまでもデバイスコンテキストから myPen を解放することです。結果としてもとのペンが復活するのは、単なるオマケにすぎないと思ってもよいくらいです。

```
pDC->SelectObject(pOldPen);        // (a)
```

ペンを削除する際には、その前にデバイスコンテキストから切り離すことを決して忘れないでください。この作業を怠ると、Windowsの動作に深刻な悪影響を及ぼします。

次に、こうしてmyPenがデバイスコンテキストから自由になったところで、myPenが管理していたペンのGDIオブジェクトをシステムメモリから消去します。

```
myPen.DeleteObject();                    // (b)
```

上の(a)と(b)は、pDC->SelectObject(pOldPen)がmyPenへのポインタを返すことを利用して、次のように1行にまとめて書くことも可能です。しかしプログラムが読みにくくなるので、あまりお勧めはしません。

```
(pDC->SelectObject(pOldPen))->DeleteObject();
```

以上の手順でペンを作成し、クライアント領域に線を引く実例が、G1プロジェクトのG1View.cppに含まれるPenTest1関数です。**リスト1-3**にこの関数のコードを示します。

PenTest1関数はOnDraw関数から呼び出されます。このとき引数pDCにクライアント領域のデバイスコンテキストへのポインタを渡しています。

リスト1-3　ペンの作成と利用のテスト：PenTest1関数(G1View.cpp)

```
void PenTest1(CDC* pDC)    // CreatePen関数でペンを作る方法
{
  CPen  myPen;
  CPen* pOldPen;

  myPen.CreatePen(PS_SOLID, 1, RGB(255, 0, 0)); // ペンを作成
  pOldPen = pDC->SelectObject(&myPen);          // 選択
  pDC->MoveTo(10, 20);                          // 線を引く
  pDC->LineTo(210, 20);
  pDC->SelectObject(pOldPen);                   // 選択終了
  myPen.DeleteObject();                         // ペンを削除
}
```

さきほどの空のPenTest1関数にこのコードを記述して、プログラムをコンパイル／実行すると、画面の最上部に水平の赤い直線が引かれることを確かめてください(**図1-2**)。

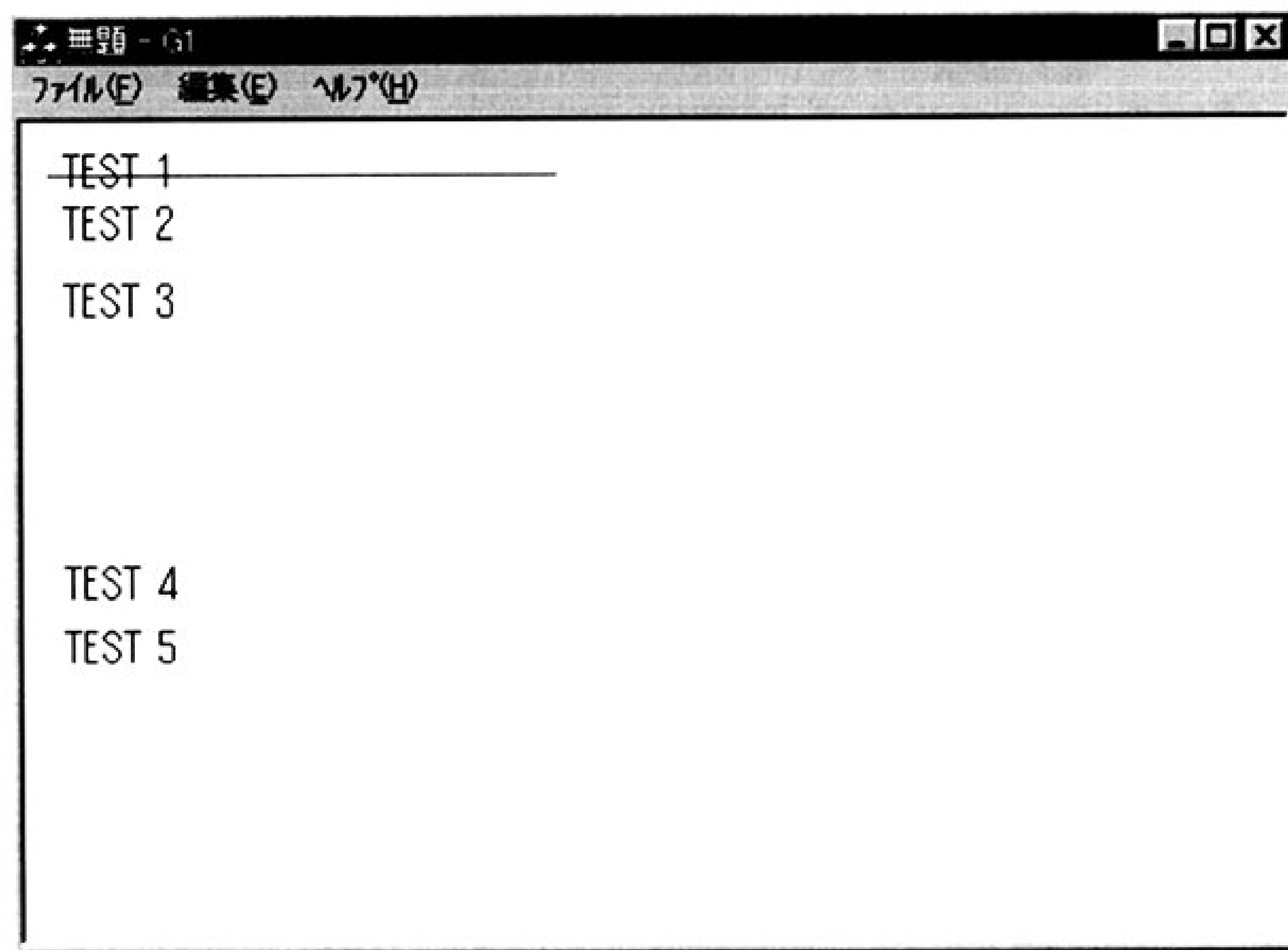


図 1-2 PenTest1 の実行結果

ペンのスタイル、太さ、色

CPen::CreatePen 関数の引数は、**図 1-3** のように、作成するペンのスタイル、太さ、および色を指定します。

CreatePen(int nPenStyle, int nWidth, COLORREF crColor);

ペンのスタイル　ペンの太さ　ペンの色

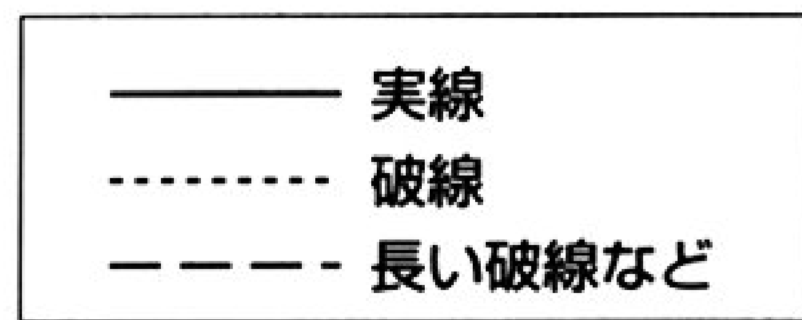


図 1-3 ペンの属性の指定

第 1 引数には**表 1-2** に示すスタイルを指定します。ここで PS_NULL と PS_INSIDE FRAME の 2 つは、主に長方形や楕円などの枠として利用するペンスタイルです。

PS_SOLID スタイルと PS_INSIDEFRAME スタイルでは、CPen::CreatePen 関数の第 2 引数によって線の太さを指定できます。そのほかのペンスタイルでは 1 以外の太さは選べません（点線タイプのペンに 1 以外の太さを指定すると、強制的に実線に変更されます）。

CPen::CreatePen 関数の第 3 引数はペンの色を指定します。Windows プログラムでは、すべての色情報を**図 1-4** のような 24 ビットの論理カラー（COLORREF 型データ）で表します。rgb の各成分は、それぞれ 8 ビットの範囲の整数値（0～255）です。

ペンスタイル	線の形状	太さ指定	論理カラー変換法
PS_SOLID	実線	○可能	純色化
PS_DOT	短い破線	×不可	純色化
PS_DASH	長い破線	×不可	純色化
PS_DASHDOT	1点鎖線	×不可	純色化
PS_DASHDOTDOT	2点鎖線	×不可	純色化
PS_NULL	透明	×不可	純色化
PS_INSIDEFRAME	実線(枠の境界内に収まる)	○可能	ディザリング注

注：PS_INSIDEFRAMEスタイルでもペンの太さが1の場合は純色化される

表1-2　ペンスタイル



図1-4　論理カラー(COLORREF型データ)

RGB(r, g, b)というマクロを使うと、rgb成分によって論理カラーが指定できます。たとえばRGB(255, 0, 0)は、赤成分255、緑成分0、青成分0ですから、純粋な赤ということになります。

論理カラーは全体で2の24乗(約1670万)色が表現可能ですが、ハードウェアの制限でそれほど多くの色を出せない場合は、実際の表示の際に論理カラーを適当な色に変換します。変換方法には次の2種類があり、Windowsは状況によって両者を使い分けます(**表1-3**)。

変換方法	論理カラーと実際の表示色の対応
純色化	そのハードウェアで表示可能な色(純色)の中で、指定した論理カラーにもっとも近いものを選ぶ
ディザリング	異なる色の点を交互に配置して色を混ぜ、指定した論理カラーを擬似的に作り出す

表1-3　色変換の方法

ペンによる線の描画の際は、基本的には純色化によって、論理カラーから実際の表示色への変換が行われます。ただしPS_INSIDEFRAMEスタイルで1以外の太さを指定した場合には、ディザリングが用いられます。

コンストラクタとデストラクタの利用

コンストラクタは、クラスのオブジェクトを定義したときに、かならず呼び出される関数で、この中でクラスのメンバの初期化などの作業を行います。コンストラクタはクラスと同じ名前のメンバ関数として定義され、たとえばCPenクラスのコンストラクタの名前はCPen::CPen()となります。

CPenクラスには、次の2種類のコンストラクタが用意されています。C++言語では、引数の並びが違う場合には同名の関数を複数定義することが可能であり、これを**関数のオーバーロード**と呼びますが(詳細はAppendix Aを参照)、CPenクラスでは関数のオーバーロードを利用して、以下の2つのコンストラクタを提供しているのです。

```
CPen::CPen();
CPen::CPen(int PenStyle, int PenWidth, COLORREF color);
```

引数を持たない1つ目のコンストラクタは、単にCPenクラスのオブジェクトを作成するだけで、明示的にCPen::CreatePen関数を実行しないとペンのGDIオブジェクトは作成されません。それに対して引数を持つ2つ目のコンストラクタは、CPenクラスのオブジェクトを確保した上に、ペンのGDIオブジェクトの作成まで行うように定義されています。

たとえば前項のプログラムでは、次のように引数を指定せずにmyPenを定義しました。この場合は引数を持たないコンストラクタが使用されます。

```
CPen   myPen;      //引数を持たないコンストラクタが起動
```

しかしmyPenを定義する際に次のように引数を指定すると、引数を持つコンストラクタが使用され、CPenクラスのオブジェクトの確保と同時にペンのGDIオブジェクトを作成することができます。

```
CPen   myPen(PS_SOLID, 1, RGB(0, 0, 255)); //引数を持つコンストラクタが起動
```

つまりこれは、以下の手順によるmyPenの作成と同じことです。

```
CPen   myPen;
myPen.CreatePen(PS_SOLID, 1, RGB(0, 0, 255));
```

コンストラクタがオブジェクト作成時に実行されるのに対し、オブジェクトが解放されるときにかならず実行される関数がデストラクタです。デストラクタはクラス名の先頭に「~」(チルダ)を付けた名前のメンバ関数として定義され、オブジェクトの解放に伴うさまざまな後始末を行います。たとえばCPenクラスのデストラクタCPen::~CPen関数には、自動的にペンのGDIオブジェクトを削除する機能が与えられています。

ということは、前項のプログラムで最後に実行していたCPen::DeleteObject関数は、省略することができるわけです。myPenというCPenクラスのオブジェクトを利用していた関数が終了する時点で、CPenクラスのデストラクタが実行され、CPen::DeleteObject関数が自動的に実行されるからです（デストラクタに頼らず明示的にDeleteObject関数を実行しても害はない）。

リスト1-4に示すPenTest2関数は、引数を持つコンストラクタを利用してペンを作成し、デストラクタを利用してペンの削除を行います。

リスト1-4 引数を持つコンストラクタの利用：PenTest2関数（G1View.cpp）

```
void PenTest2(CDC* pDC) //コンストラクタでペンを作ると簡単
{
  CPen  myPen(PS_SOLID, 1, RGB(0, 0, 255));   //ペンを作成
  CPen* pOldPen;

  pOldPen = pDC->SelectObject(&myPen);
  pDC->MoveTo(10, 40);
  pDC->LineTo(210, 40);
  pDC->SelectObject(pOldPen);   //削除はデストラクタが行ってくれる
}
```

このPenTest2関数と、さきほどのPenTest1関数を比べてみてください。どちらもほとんど同じ作業をしているのですが、引数付きのコンストラクタやデストラクタにペンの作成と削除をまかせたおかげで、PenTest2関数はプログラムがPenTest1関数よりもすっきりしたものとなっています。

というわけで、引数を取るコンストラクタを使うと、タイピングの手間は省けるしプログラムの見通しもよくなるし、こんな結構なものはありません。とくに、PenTest2関数のように関数の入口で定型のペンを作成し、関数の出口でそれを削除するといった形式のプログラムでは、積極的に利用するべきです。

ただし、いつでもコンストラクタに頼っていればよいかというと、そうもいきません。PenTest3関数を見てください（**リスト1-5**）。

リスト1-5 明示的にCreatePen関数を実行した方がよい場合：PenTest3関数（G1View.cpp）

```
void PenTest3(CDC* pDC)    //だからってコンストラクタだけ覚えてもだめ
{
  CPen  myPen;
  CPen* pOldPen;
  int i;
  COLORREF cr;
```

```
  for (i = 0; i < 30; i++) {
    cr = RGB(rand() % 256, rand() % 256, rand() % 256); //乱数で色を変更
    myPen.CreatePen(PS_SOLID, 1, cr);
    pOldPen = pDC->SelectObject(&myPen);
    pDC->Arc(10+i, 60+i, 210-i, 260-i, 220, 160, 0, 160); //円弧を描画
    pDC->SelectObject(pOldPen);
    myPen.DeleteObject();
  }
}
```

PenTest3関数では、線の色を次々と変えながら、半円を30個描いています（**図1-5**）。線の色を変えるといっても、デバイスコンテキストのペンの色だけを変える簡単な方法はありませんから、結局は新しい色のペンを作り、ペン全体をまるごと交換しなければなりません。

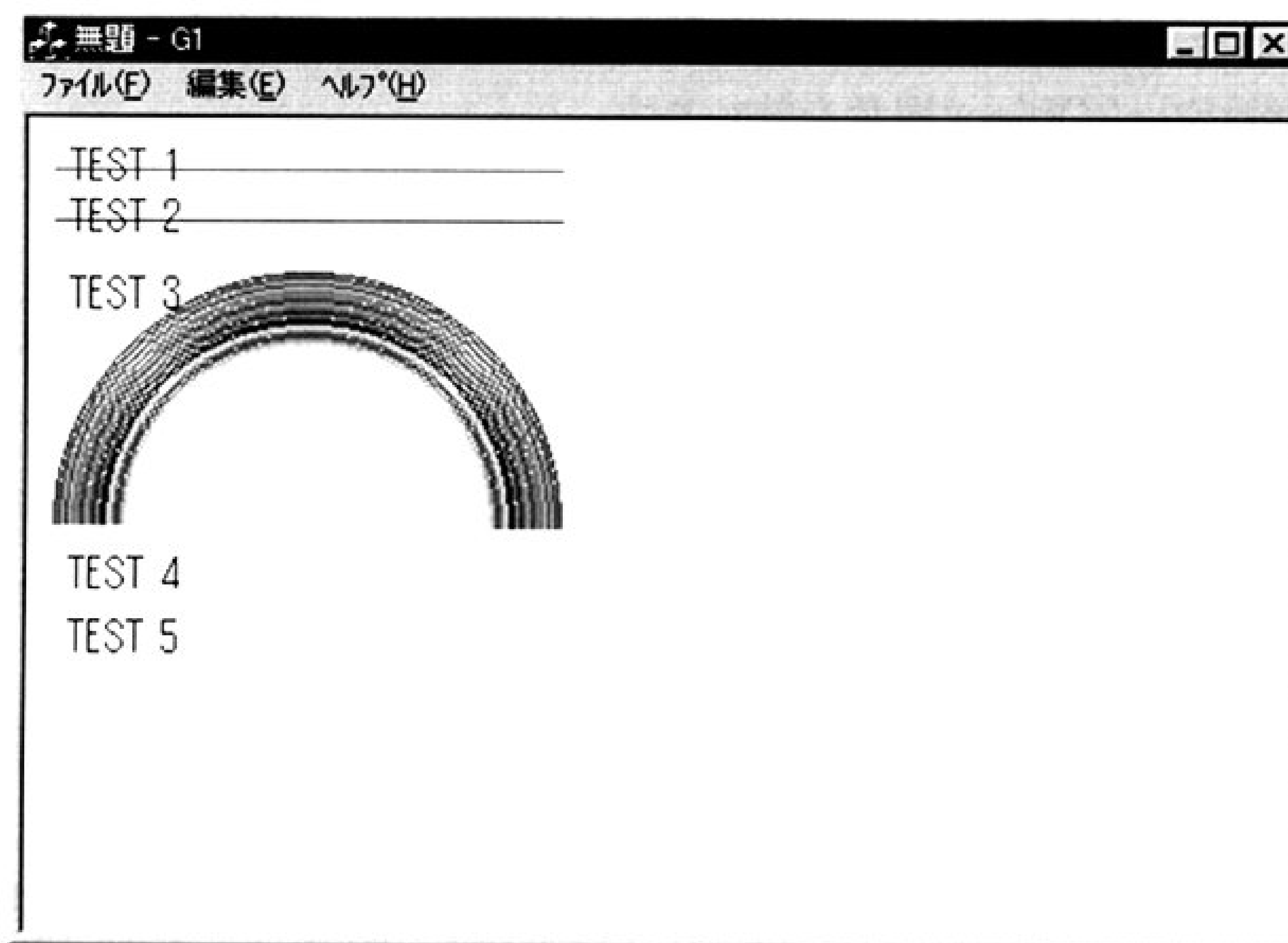


図1-5　PenTest3関数の実行結果

このような例では、やはり自分でCreatePen関数とDeleteObject関数を実行し、ペンの作成／削除をする必要があります。

ストックペンの利用

Windowsシステムには、利用頻度の高いGDIオブジェクトが最初からいくつか用意されています。これを**ストックオブジェクト**と呼びます。ペンについても、**表1-4**に示す3種類のストックオブジェクト(**ストックペン**)が利用可能です。

ストックペン	形状
BLACK_PEN	黒色の実線(太さ1:デフォルトのペン)
WHITE_PEN	白色の実線(太さ1)
NULL_PEN	透明

表1-4 ストックペンの種類

ストックペンは、いちいち作成したり削除する必要がなく、たいへん手軽に扱うことができます。デバイスコンテキストに初期設定として与えられているペンも、ストックペンの1つのBLACK_PENです。

デバイスコンテキストにストックペンを割り当てるには、CDC::SelectObject関数ではなく、CDC::SelectStockObject関数を使います。

```
pDC->SelectStockObject(WHITE_PEN);   // DCにWHITE_PENを割り当てる
```

このときCDC::SelectStockObject関数は、それまでデバイスコンテキストに割り当てられていたペンへのポインタを返します。ただし、この関数の返り値のデータ型は、GDIオブジェクト一般を意味するCGdiObjectクラスへのポインタ(CGdiObject*型)のため*2、CPenクラスへのポインタに代入するには、形式的に次のようなキャストを行う必要があります。

```
pOldPen = (CPen*) pDC->SelectStockObject(WHITE_PEN);
```

なおC++言語では、このように基底クラス(CGdiObjectクラス)へのポインタから派生クラス(CPenクラス)へのポインタに型変換を行う場合はキャストが必要ですが、その逆方向の型変換は自動的に行われます。

*2 これに対し、CDC::SelectObject関数の場合は、引数の型によって返り値の型が異なる。たとえば引数の型がCPen*型なら返り値の型もCPen*型に、引数の型がCBrush*型なら返り値の型はCBrush*型になる。これはCDC::SelectObject関数が引数の型ごとにオーバーロードされているためである。CDC::SelectStockObject関数は、int型の引数1つでストックオブジェクトを指定できるので、引数はint型が1つだけで十分である。よって、関数のオーバーロードをすることができない。そのため、CDC::SelectStockObject関数は常にCGdiObject*型を返すのである。

PenTest4関数は、ストックペンを使って線を引くサンプルです。ここまで見てきた他のプログラム（PenTest1関数～PenTest3関数）と異なり、新しいペンを作らなくともよい点に注目してください（**リスト1-6**）。

リスト1-6　ストックペンの利用例：PenTest4関数（G1View.cpp）

```
void PenTest4(CDC* pDC)    // ストックペンは作成および削除の必要がない
{
  CPen* pOldPen;

  pOldPen = (CPen*) pDC->SelectStockObject(BLACK_PEN);
  pDC->MoveTo(10, 180);
  pDC->LineTo(210, 180);
  pDC->SelectObject(pOldPen);
}
```

PenTest4関数では、最後にCDC::SelectObject関数を実行して、デバイスコンテキストからBLACK_PENを切り離しています。しかしペンが黒いままで問題ないのなら、この作業は省略してもよいでしょう。CPen::CreatePen関数で作成したペンの場合は、最後にかならずデバイスコンテキストから切り離して削除しないといけませんでしたが、ストックペンはその必要はありません。

点線の表示形式と背景描画モード

点線タイプのペン（PS_DOT、PS_DASH、PS_DASHDOT、PS_DASHDOTDOT）を使う場合は、点線のすきまの見え方にも気を使わなくてはなりません。すきまの表示形式を左右するのは、ペン自体の設定ではなく、デバイスコンテキスト側の背景描画モードです。

デバイスコンテキストの背景描画モードには**表1-5**に示す2種類があります。

背景描画モード	意味
TRANSPARENT	透明モード（背景が透けて見える）
OPAQUE	不透明モード（CDC::SetBkColor関数で指定した背景色が表示される）

表1-5　背景描画モード

PenTest5関数は、背景描画モードの違いで点線の表示がどのように変わるかを見るサンプルです（**リスト1-7**）。

リスト1-7 背景描画モードが点線の表示に与える影響：PenTest5関数（G1View.cpp）

```
void PenTest5(CDC* pDC)    // 点線タイプのペンと背景描画モードの関係
{
  CPen  myPen(PS_DOT, 1, RGB(255, 0, 0));
  CPen* pOldPen;

  pOldPen = pDC->SelectObject(&myPen);

  pDC->SetBkMode(TRANSPARENT);        // 透明モード
  pDC->MoveTo(10, 200);
  pDC->LineTo(210, 200);

  pDC->SetBkMode(OPAQUE);             // 不透明モード
  pDC->SetBkColor(RGB(0, 0, 255));    // 背景色を指定する
  pDC->MoveTo(10, 210);
  pDC->LineTo(210, 210);

  pDC->SelectObject(pOldPen);
}
```

PenTest5関数は、同じペンを使って、PS_DOTスタイルの赤い点線を2本表示するものですが、最初は次のようにデバイスコンテキストをTRANSPARENT（透明）モードに設定しています。TRANSPARENTモードでは点線のすきまに何も表示されず、背景が透けて見えます。

```
pDC->SetBkMode(TRANSPARENT);        // 背景描画モードをTRANSPARENTに設定
```

そして2本目の直線を引く前に、デバイスコンテキストをOPAQUE（不透明）モードに設定し、さらに背景色を青としています。背景色の設定にはCDC::SetBkColor関数を使用します。

```
pDC->SetBkMode(OPAQUE);             // 背景描画モードをOPAQUEに設定
pDC->SetBkColor(RGB(0, 0, 255));    // 背景色を青に設定
```

OPAQUEモードでは、CDC::SetBkColor関数で指定した色で背景が塗られます。結果として、この場合はペン自体の赤色と背景の青色が交互に並んだ点線が表示されます。最後にPenTest5の実行結果を示しておきます（**図1-6**）。

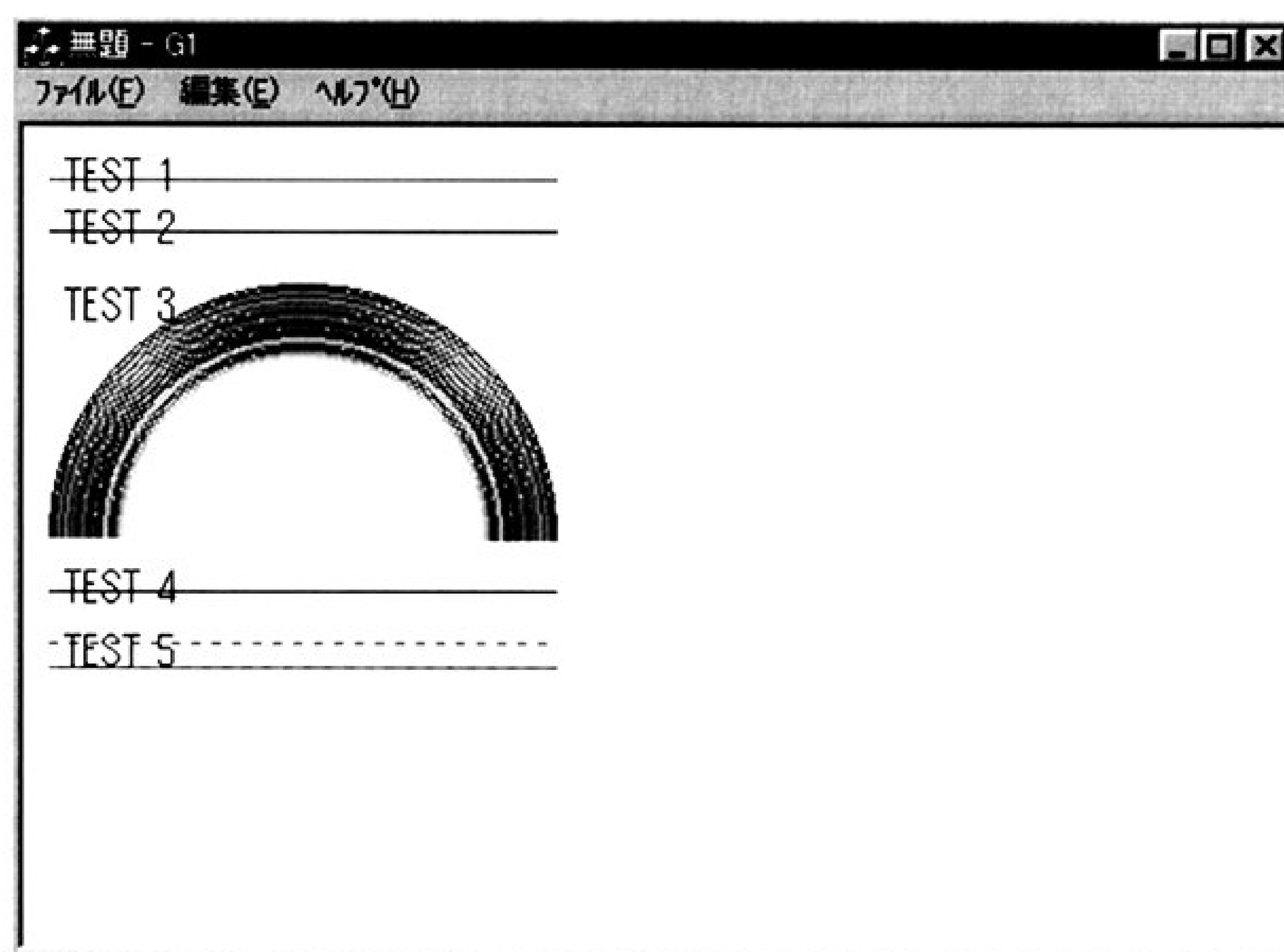


図 1-6 PenTest5 関数の実行結果

● ブラシの利用法 ―― 図形の塗りつぶしスタイルを指定する

ブラシは CDC::Rectangle 関数や CDC::Ellipse 関数で矩形や楕円を描くときに、その内部を塗りつぶすのに利用される GDI オブジェクトです。Windows ではさまざまなブラシが利用できますが、ここでは単色のソリッドブラシと、単純な線パターンからなるハッチブラシの使い方を説明します。

G2 プロジェクトの設計

それでは、ペンの GDI オブジェクトの実際を見ていく前に、テストプログラムを記述するためのプロジェクトを作成しておきましょう。以下は AppWizard と、そのオプションで定めるアプリケーションの枠組みです。

- プロジェクト名：G2
- アプリケーションのタイプ：SDI
- データベースのサポート：しない
- 複合ドキュメントのサポート：しない
- ツールバー／ステータスバー：なし
- 印刷と印刷プレビュー：なし
- そのほか：デフォルトのまま

上の方針に従って、AppWizard でプロジェクトを新規に開始してください。そしてスケルトンが完成したら、G1 プロジェクトのときと同様に、G2View.cpp の OnDraw 関数の辺りを**リスト 1-8** のように修正してください。BrushTest1～BrushTest3 の 3 つの関数

は、それぞれOnDraw関数から呼び出されて、ブラシを使った簡単なテストを行うものですが、ここではとりあえず関数の入口だけ用意しておきます。

リスト1-8 ブラシテストの前準備(G2View.cpp)

```
////////////////////////////////////////
// CG2Viewクラスの描画
void BrushTest1(CDC* pDC)
{
}

void BrushTest2(CDC* pDC)
{
}

void BrushTest3(CDC* pDC)
{
}

void CG2View::OnDraw(CDC* pDC)
{
  pDC->TextOut(16,   20, "TEST 1");  BrushTest1(pDC);
  pDC->TextOut(16,  180, "TEST 2");  BrushTest2(pDC);
  pDC->TextOut(216, 100, "TEST 3");  BrushTest3(pDC);
}
```

ブラシの作成

ブラシのGDIオブジェクトはCBrushクラスで管理します。ブラシを利用するには、まず最初にCBrushクラスのオブジェクトを定義します。

```
CBrush  myBrush;
```

そしてGDIオブジェクトのブラシを作成し、このオブジェクトと結び付けます。ブラシを作成する関数は、ペンの場合と異なり、ブラシの種類ごとに用意されています。

ソリッドブラシはCBrush::CreateSolidBrush関数を使って作成します。この関数は引数でブラシの論理カラーを指定します。たとえば次の例では赤い単色のブラシが作成されます。

```
myBrush.CreateSolidBrush(RGB(255, 0, 0)); // ソリッドブラシ
```

なおソリッドブラシの表示の際には、論理カラーに対してディザリングが行われるので、256色しか表示できないハードウェアでも擬似的に1670万色が表現できます。

ハッチブラシの作成には CBrush::CreateHatchBrush 関数を使用します。この関数は第 1 引数でハッチパターンを指定し、第 2 引数で論理カラーを指定します。次の例は HS_CROSS 形式（縦横の格子模様）の青いハッチブラシを作るものです。

```
myBrush.CreateHatchBrush(HS_CROSS, RGB(0, 0, 255)); // ハッチブラシ
```

ハッチパターンは、**表 1-6** に示す 6 種類があります。

ハッチパターン	形状
HS_HORIZONTAL	横線
HS_VERTICAL	縦線
HS_BDIAGONAL	右上がりの斜め線
HS_FDIAGONAL	左上がりの斜め線
HS_CROSS	縦横の格子
HS_DIAGCROSS	斜めの格子

表 1-6 ハッチパターンの種類

ハッチブラシを作成する場合は、引数に指定した論理カラーが純色化されて表示されます。つまり実際のハッチブラシの色は、ハードウェアの表示能力に制限されるということです。

さて、作成したブラシは、ペンと同様に CDC::SelectObject 関数を実行し、デバイスコンテキストに割り当てますが、このとき SelectObject 関数は、デバイスコンテキストがそれまで使用していたブラシへのポインタを返します。

```
CBrush* pOldBrush;
pOldBrush = pDC->SelectObject(&myBrush);
```

前項では、ペンの選択時に CDC::SelectObject 関数が CPen*型の戻り値を返すことを説明しました。しかしブラシを選択する場合は、CDC::SelectObject 関数の戻り値は CBrush*型になるのです。これも C++言語の関数のオーバーロードを利用したものです。

デバイスコンテキストにブラシを割り当てると、以後はそのブラシによって図形の内部が塗りつぶされるようになります。こうして描画を終えて用済みになったブラシは、ペンと同様に、システムメモリから削除してください。

```
pDC->SelectObject(pOldBrush); // myBrush を切り離す
myBrush.DeleteObject();       // ブラシの GDI オブジェクトを削除する
```

リスト 1-9 は実際にブラシを利用したプログラム例です。BrushTest1 関数は、赤いソリッドブラシで塗られた円と、青のハッチブラシで塗られた円を表示します（**図 1-7**）。

リスト 1-9 ブラシの作成と利用：BrushTest1 関数（G2View.cpp）

```
void BrushTest1(CDC* pDC)   // ブラシの作成と削除
{
  CBrush  myBrush;
  CBrush* pOldBrush;

  myBrush.CreateSolidBrush(RGB(255, 0, 0));   // ソリッドブラシの作成
  pOldBrush = pDC->SelectObject(&myBrush);    // myBrush を選択
  pDC->Ellipse(60, 10, 160, 110);
  pDC->SelectObject(pOldBrush);               // もとに戻す
  myBrush.DeleteObject();                     // ブラシを削除

  myBrush.CreateHatchBrush(HS_CROSS, RGB(0, 0, 255)); // ハッチブラシの作成
  pOldBrush = pDC->SelectObject(&myBrush);    // myBrush を選択
  pDC->Ellipse(110, 10, 210, 110);
  pDC->SelectObject(pOldBrush);               // もとに戻す
  myBrush.DeleteObject();                     // ブラシを削除
}
```

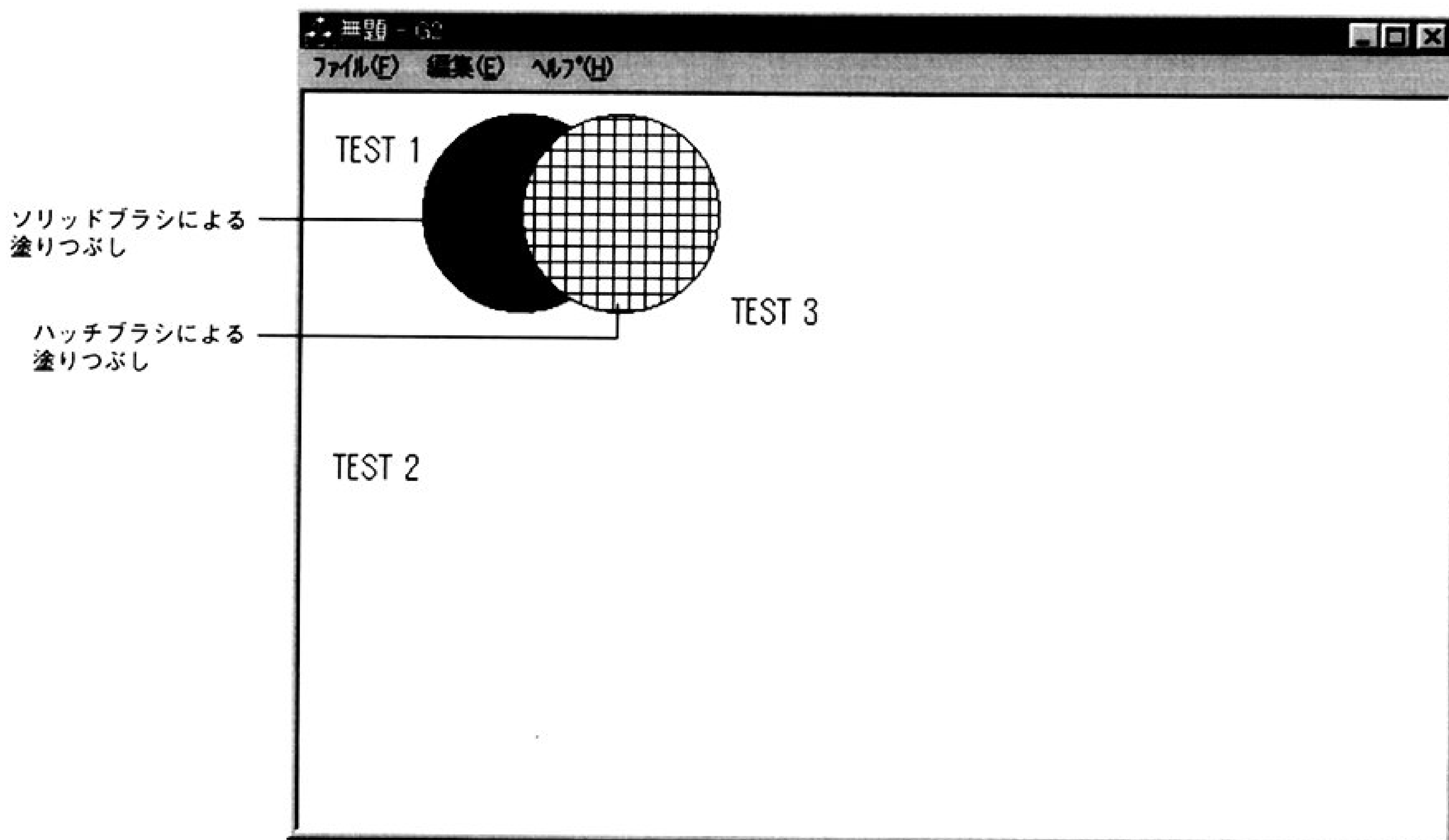


図 1-7 BrushTest1 関数の実行画面

引数を持つコンストラクタ

CBrush クラスでは、コンストラクタに与える引数によって、異なる種類のブラシを作ることができます。ソリッドブラシを作成するには、次のように論理カラーを唯一の引数としてオブジェクトを定義します。

```
CBrush  myBrush(RGB(255, 255, 0));              //ソリッドブラシ
```

ハッチブラシを作成するには、次のように第1引数にハッチパターン、第2引数に論理カラーを指定して変数を定義します。

```
CBrush  myBrush(HS_DIAGCROSS, RGB(0, 255, 0));  //ハッチブラシ
```

ストックブラシの利用

ストックオブジェクトとして Windows システム内にあらかじめ用意されているブラシのことを**ストックブラシ**と呼びます。ストックブラシには**表 1-7** に示すものがあります。

ストックブラシ	形状	論理カラー
BLACK_BRUSH	黒色のソリッドブラシ	RGB(0, 0, 0)
DKGRAY_BRUSH	暗い灰色のソリッドブラシ	RGB(64, 64, 64)
GRAY_BRUSH	灰色のソリッドブラシ	RGB(128, 128, 128)
LTGRAY_BRUSH	明るい灰色のソリッドブラシ	RGB(192, 192, 192)
WHITE_BRUSH	白色のソリッドブラシ(デフォルトのブラシ)	RGB(255, 255, 255)
NULL_BRUSH	透明(ヌルブラシ)	なし
HOLLOW_BRUSH	NULL_BRUSH の別名	なし

表 1-7 ストックブラシの種類

デフォルトの状態では、デバイスコンテキストには WHITE_BRUSH が与えられています。ストックブラシをデバイスコンテキストに割り当てるには、次のように CDC::SelectStockObject 関数を使ってください。

```
pOldBrush = (CBrush*) pDC->SelectStockObject(NULL_BRUSH);
```

ストックブラシの中でも重要なものが、NULL_BRUSH(または HOLLOW_BRUSH)で表される**ヌルブラシ**です。ヌルブラシは透明色ブラシ、つまり画面に何も塗らないブラシです。図形の枠だけを描く場合にはヌルブラシを使います。

リスト 1-10 にヌルブラシを使うサンプルを示します。この BrushTest2 関数を実行すると円の枠だけが描かれることを確認してください(**図 1-8**)。

リスト1-10　ヌルブラシを使う：BrushTest2関数(G2View.cpp)

```
void BrushTest2(CDC* pDC)    // ヌルブラシは塗らない
{
  CBrush* pOldBrush;

  pOldBrush = (CBrush*) pDC->SelectStockObject(NULL_BRUSH);
  pDC->Ellipse(60, 170, 160, 270);
  pDC->Ellipse(110, 170, 210, 270);
  pDC->SelectObject(pOldBrush);
}
```

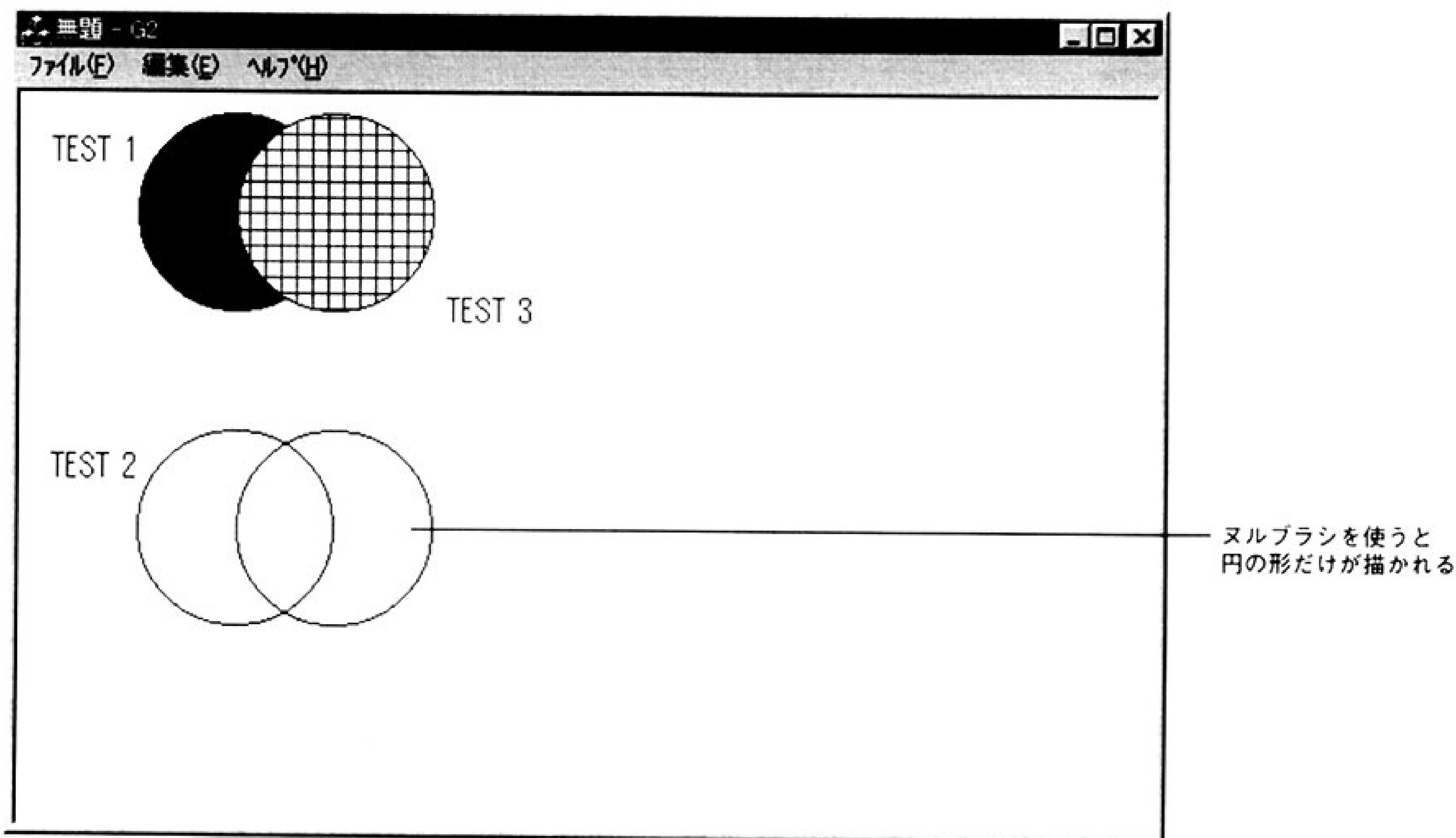


図1-8　BrushTest2関数の実行結果

ハッチブラシと背景描画モード

ハッチブラシのパターンのすきまは、点線のすきまの場合と同様に、デバイスコンテキストの背景描画モードによって表示様式が変わります。

デバイスコンテキストをOPAQUEモードに設定すると、ハッチブラシのすきまは背景色で塗りつぶされます。ただしこのとき背景色は純色化されます。一方、デバイスコンテキストをTRANSPARENTモードにすると、ハッチブラシの直線パターンのみが表示され、パターンのすきまは透明になります。

リスト1-11は、背景描画モードの設定がハッチブラシに与える影響を見るサンプルです。TRANSPARENTモードでは下の絵が透けて見えるところに注目してください(**図1-9**)。

リスト 1-11　背景描画モードがハッチブラシに与える影響：BrushTest3 関数（G2View.cpp）

```
void BrushTest3(CDC* pDC)    // ハッチブラシと背景描画モードの関係
{
  CBrush  myBrush1(HS_FDIAGONAL, RGB(0, 0, 0));
  CBrush  myBrush2(HS_BDIAGONAL, RGB(0, 0, 0));
  CBrush* pOldBrush;

  pDC->SetBkMode(OPAQUE);                   // OPAQUE モードでは
  pDC->SetBkColor(RGB(0, 255, 255));        // 線の間が背景色で塗られる
  pOldBrush = pDC->SelectObject(&myBrush1);
  pDC->Ellipse(260, 100, 360, 180);

  pDC->SetBkMode(TRANSPARENT);              // TRANSPARENT モードは
  pDC->SelectObject(&myBrush2);             // 線の間が透けて見える
  pDC->Ellipse(310, 100, 410, 180);

  pDC->SelectObject(pOldBrush);
}
```

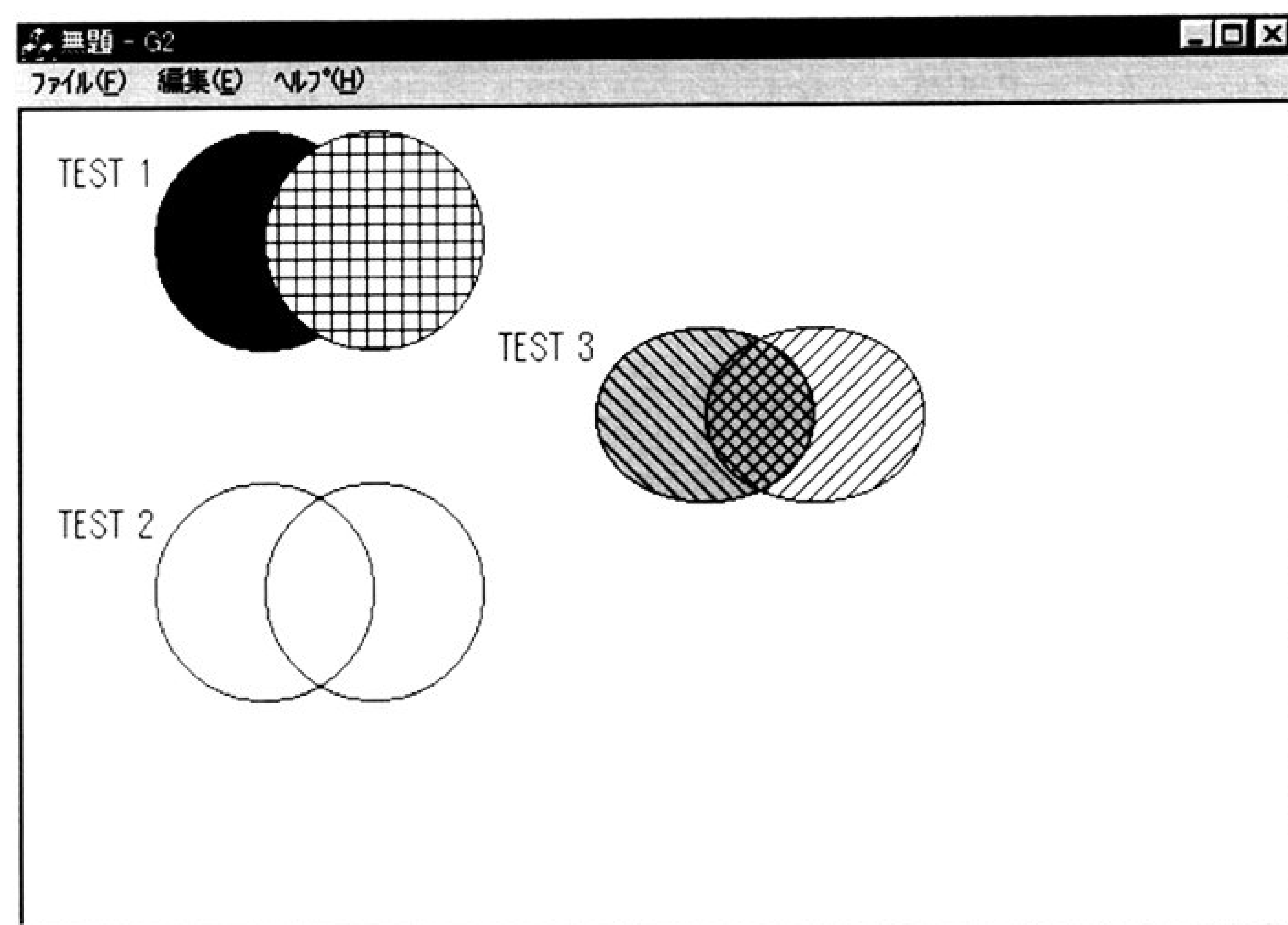


図 1-9　BrushTest3 関数の実行結果

●ビットマップの利用法 ―― 任意図形の表示

ビットマップの場合は、ペンやブラシよりも利用手順が少々複雑です。まずビットマップの図形データを用意しなければなりません。またビットマップを画面に表示するためには、メモリデバイスコンテキストの利用方法や、ビットブリット（BitBlt：ビットブロック転送命令）の使い方も覚える必要があります。

G3プロジェクトの設計とリソースの準備

G3プロジェクトではビットマップ図形をクライアント領域に表示するプログラムを作成します。プログラムの動作は、OnDraw関数を利用して、あらかじめ用意したビットマップ図形をクライアント領域一杯に敷き詰めて表示するだけの簡単なものです。

まずAppWizardを起動して、以下のオプション設定でスケルトンプログラムを作成してください。

- プロジェクト名：G3
- アプリケーションのタイプ：SDI
- データベースのサポート：しない
- 複合ドキュメントのサポート：しない
- ツールバー／ステータスバー：なし
- 印刷と印刷プレビュー：なし
- そのほか：デフォルトのまま

ビットマップ図形のデータを用意する方法はいくつかありますが、その中で一番簡単なのは、ビットマップリソースを利用する方法です。Developer Studio内に含まれているグラフィックエディタを使えば手作業でビットマップリソースを作成することもできますが、こ

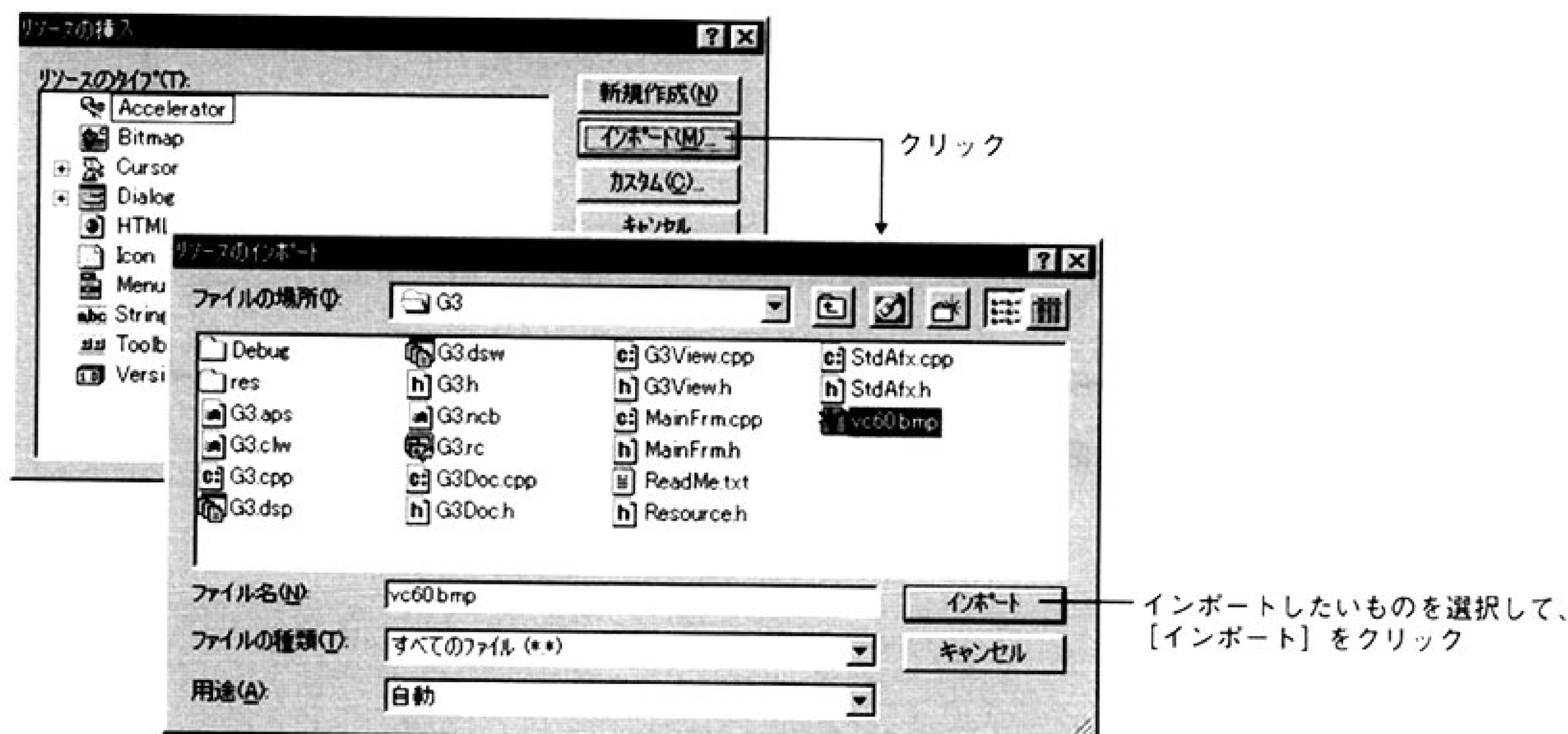


図1-10 ［リソースの挿入］ダイアログボックス

こでは付属CD-ROMのG3プロジェクト内にある、vc60.bmpを利用しました。実際に、本プログラムを作成する場合には、このビットマップを流用するか、あるいは適当なものを使用するようにしてください。

メニューから[挿入]-[リソース]を選択すると、[リソースの挿入]ダイアログボックスが表示されるので、<インポート>ボタンをクリックしてください。次に[リソースのインポート]ダイアログボックスで、[ファイルの種類]に[すべてのファイル(*.*)]を指定して、望みのファイル(ここではvc60.bmp)を選択し、<インポート>ボタンをクリックすれば、ビットマップファイルをインポートすることができます(**図1-10**)。

<インポート>ボタンをクリックすると、画面にはグラフィックエディタが現れ、そこに指定したファイルからデータが読み込まれ、ビットマップが表示されます(**図1-11**)。

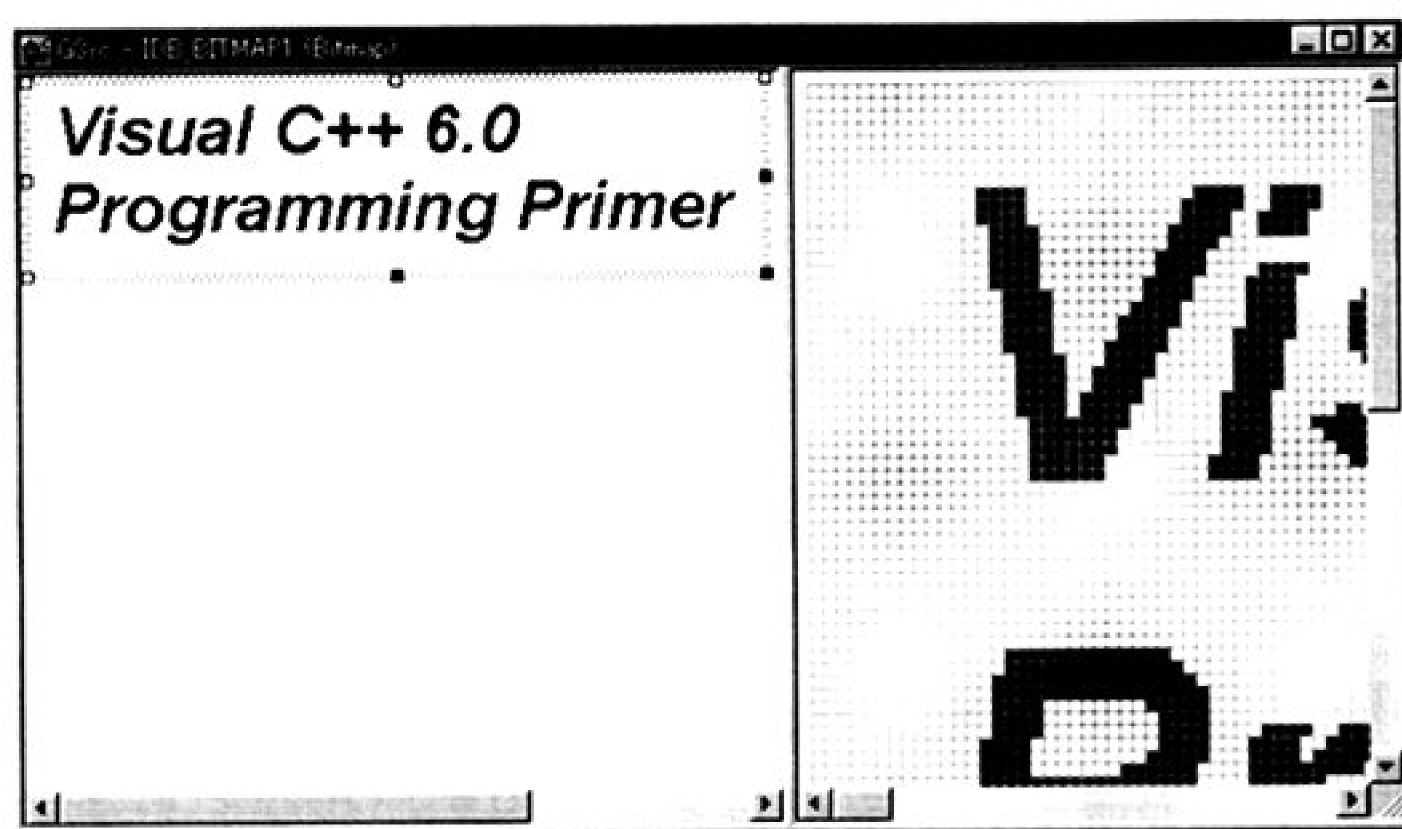


図1-11 グラフィックエディタの画面

グラフィックエディタの画面(ただし表示されているビットマップ図形の外側)をダブルクリックすると、**図1-12**のようなプロパティボックスが開きます。

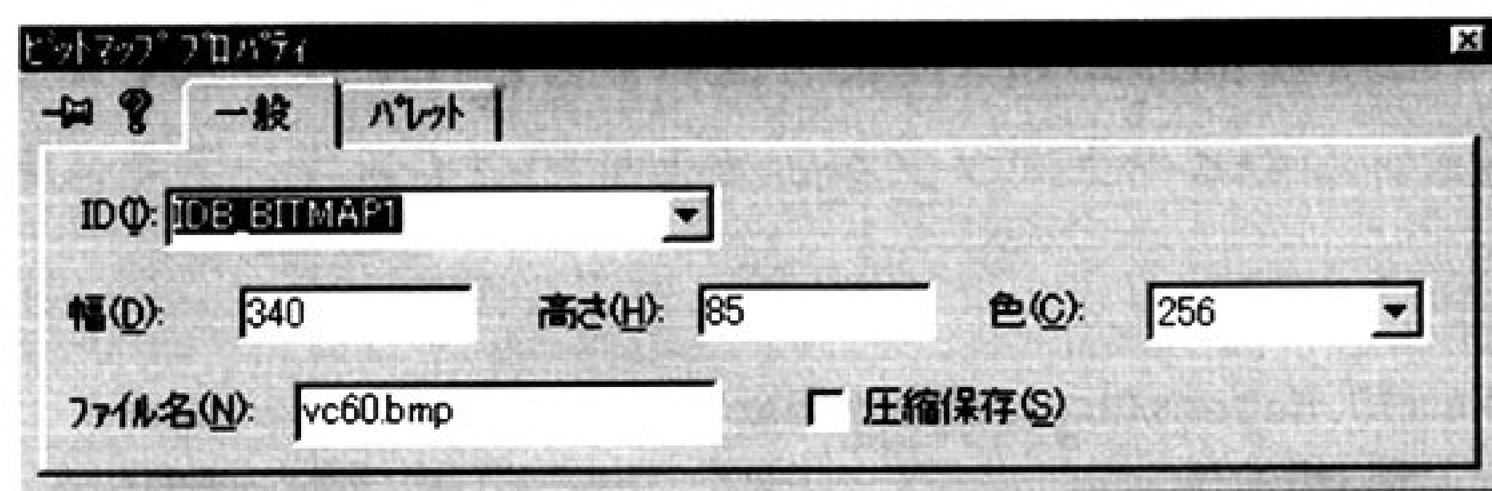


図1-12 プロパティボックス

プロパティボックスには、ビットマップのIDやサイズが表示されています。ここでは[ID]ボックスに表示されたビットマップのリソースIDを「IDB_SAMPLEBMP」に変更してください。以上でビットマップリソースの準備は完了です。

ビットマップの表示

ビットマップリソースから、CBitmapクラスのオブジェクトを作成するには、CBitmap::LoadBitmap関数を利用します。この関数は引数に指定されたビットマップリソースID（リソースエディタで指定したもの）からビットマップを作成するものです。

```
CBitmap Bitmap;
Bitmap.LoadBitmap(IDB_SAMPLEBMP);
```

ビットマップを画面に表示する方法は、他のGDIオブジェクトを画面に表示する方法とは少々異なります。表示したいデバイスコンテキストに直接そのビットマップを割り当てるのではなく（SelectObject関数を実行するのではなく）、**メモリデバイスコンテキスト**という特殊なデバイスコンテキストをデータ転送のサポート役として利用し、まずはこのサポート役にビットマップを割り当てるのです。

メモリデバイスコンテキストとは、メモリ内部に設けられた擬似的な画面のデバイスコンテキストのことです。ここでは画面バッファの役割を果たす存在だと思ってもらえれば間違いないでしょう。メモリデバイスコンテキストを利用するには、まずCDCクラスのオブジェクトを1つ用意します。

```
CDC MemDC;
```

そしてこのMemDCに対し、CDC::CreateCompatibleDC関数を実行します。この関数はデバイスコンテキストへのポインタを引数とし、引数に指定したデバイスコンテキストと同じ表示能力（主に発色能力）を備えたメモリデバイスコンテキストを作成します。

```
MemDC.CreateCompatibleDC(pDC);
```

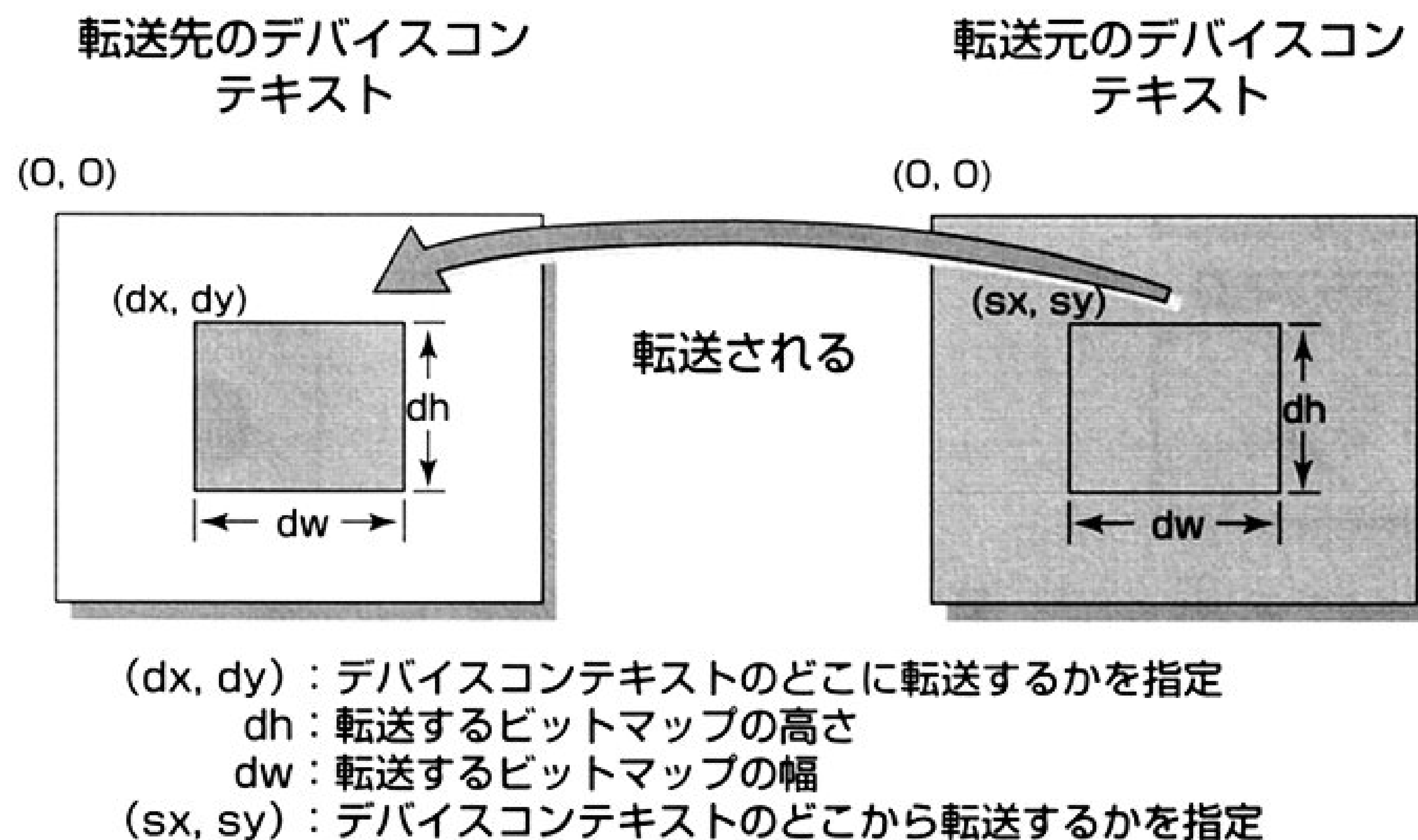
ここまでの段階では、メモリデバイスコンテキストは画面に相当するデータバッファを持っていません。そこでさきほどビットマップリソースを読み込んで作ったビットマップオブジェクトを、CDC::SelectObject関数を使ってこのメモリデバイスコンテキストに割り当てます。すると、このビットマップが、メモリデバイスコンテキストの画面になるわけです。

```
pOldBitmap = MemDC.SelectObject(&Bitmap);
```

こうして完成したメモリデバイスコンテキストは、ディスプレイ画面のデバイスコンテキストを扱うかのように、データを出力することもできれば、データを読み取ることもできます。ただしその結果は当然ながらメモリ中のデータとして記録されるだけで、目に見える変化は何も生じません。

そこで、CDC::BitBlt 関数という関数が登場します。この関数は、**図 1-13** に示すように、転送元のデバイスコンテキスト上の任意の矩形データを転送先のデバイスコンテキストの任意の位置に転送するものです。

```
pDC->BitBlt(dx, dy, dw, dh, pSrcDC, sx, sy, rop);
```



(dx, dy)：デバイスコンテキストのどこに転送するかを指定
dh：転送するビットマップの高さ
dw：転送するビットマップの幅
(sx, sy)：デバイスコンテキストのどこから転送するかを指定

図 1-13 ビットブリットの転送元と転送先の座標指定

CDC::BitBlt 関数の第 8 引数は、転送元のデータと転送先のデータの間で行われるラスタオペレーション（ビットごとの論理演算）の種類を定めます。指定できるラスタオペレーションの中で代表的なものを**表 1-8** に示します。また、ラスタオペレーションを行うとどんな結果になるかを**図 1-14** に示します。

CDC::BitBlt 関数を利用して、メモリデバイスコンテキストからディスプレイ画面のデバイスコンテキストにグラフィックデータを転送することで、ようやく目的のビットマップがディスプレイに表示されます。

また、アプリケーションのクライアント領域のサイズを得るには CWnd::GetClientRect

ラスタオペレーション	論理演算	意味
SRCCOPY	dst = src	転送元のパターンを上書き
NOTSRCCOPY	dst = ~src	転送元のパターンを反転して上書き
SRCPAINT	dst = dst | src	転送元と転送先のパターンの OR を取る
SRCAND	dst = dst & src	転送元と転送先のパターンの AND を取る
SRCINVERT	dst = dst ^ src	転送元と転送先のパターンの XOR を取る
SRCERASE	dst = ~dst & src	転送元のパターンを転送先のパターンから消去

表 1-8 代表的なラスタオペレーション

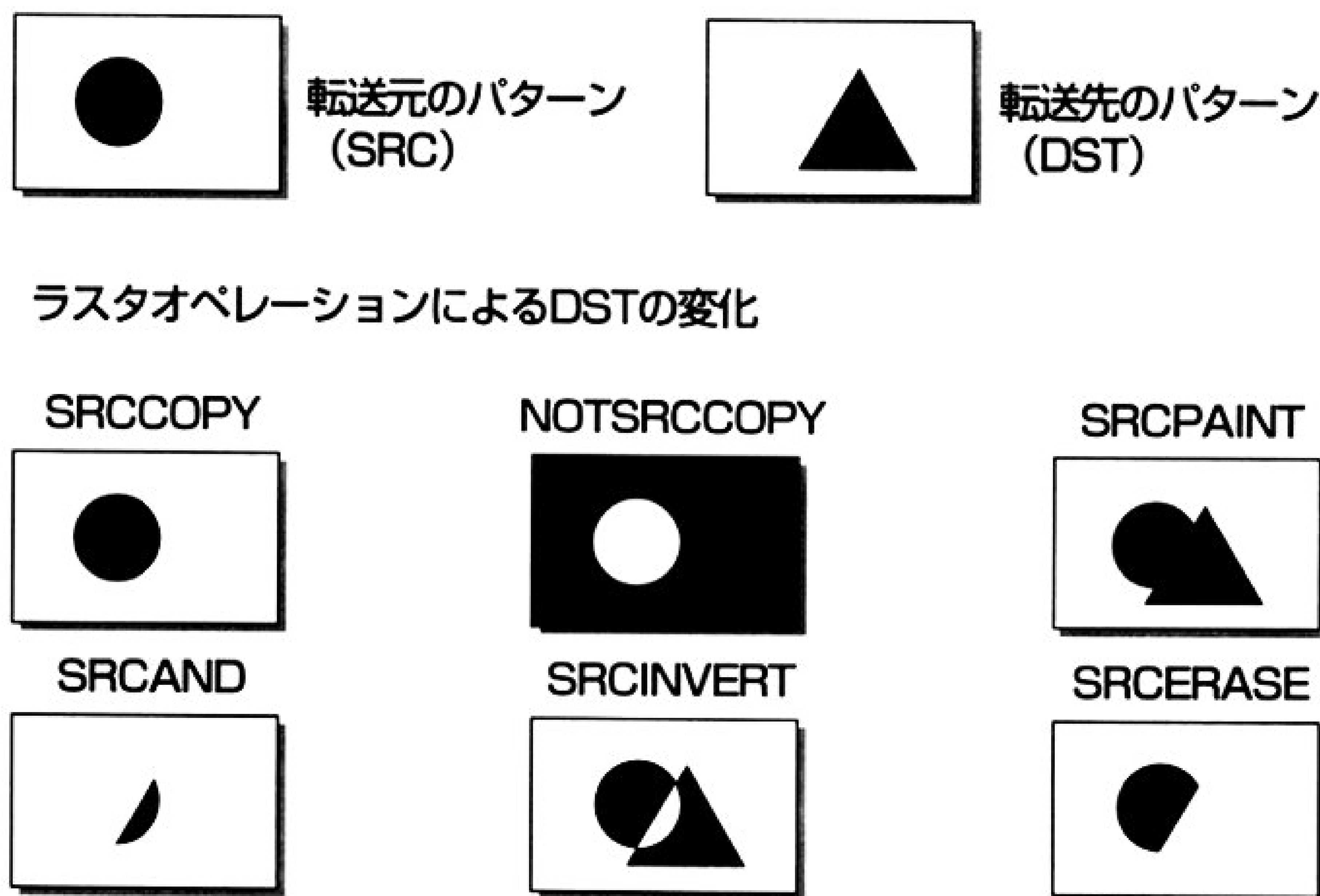


図 1-14　ラスタオペレーション

関数を使います。この関数はウィンドウのクライアント領域の座標を引数に指定した CRect クラス（または RECT 構造体）のオブジェクトにセットします。CRect クラス／RECT 構造体は共にある 2 点で決定する矩形領域の位置とそのサイズを扱うのに利用されます。

ここで説明したことを組み合わせると、ビットマップを表示する OnDraw 関数は**リスト 1-12** のようなプログラムになります。

リスト 1-12　ビットマップを転送するテスト（G3View.cpp）

```
void CG3View::OnDraw(CDC* pDC)
{
  CBitmap Bitmap, *pOldBitmap;
  CDC MemDC;      // メモリデバイスコンテキストの用意
  int x, y;
  CRect rect;

  Bitmap.LoadBitmap(IDB_SAMPLEBMP);
          // リソースエディタで作成したビットマップから CBitmap オブジェクトを作成
  MemDC.CreateCompatibleDC(pDC);
  pOldBitmap = MemDC.SelectObject(&Bitmap); // ビットマップを割り当てる
  GetClientRect(&rect); // ウィンドウのサイズを rect に記憶する
  for (y = 0; y < rect.bottom; y += 85) { // ビットマップの高さは 85 ドット
    for (x = 0; x < rect.right; x += 340) { // ビットマップの幅は 340 ドット
      pDC->BitBlt(x, y, 340, 85, &MemDC, 0, 0, SRCCOPY);  // ビットマップの転送
    }
  }
  MemDC.SelectObject(pOldBitmap);
}
```

リスト1-12に示すようにOnDraw関数を書き換えて、プロジェクトをコンパイル／実行すると**図1-15**のようにビットマップがウィンドウ全体に表示されます。

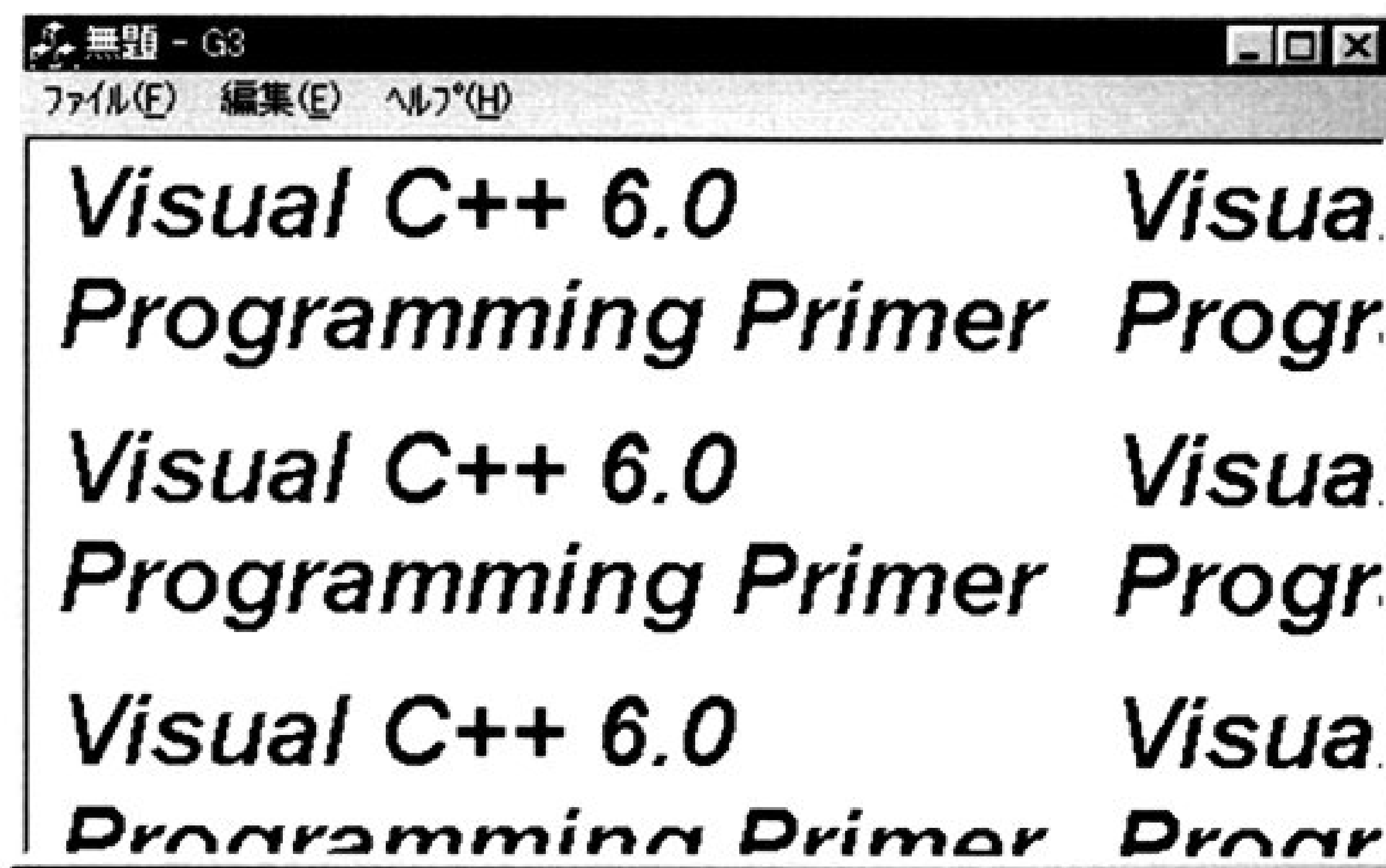


図1-15　実行画面

2 メニューを使ってみよう

GDIは画面出力に関する統一インターフェイスを実現するものでしたが、Windowsはユーザーがアプリケーションを操作するための統一的なユーザーインターフェイスも備えています。その中核となるものがメニューとダイアログボックスです。そこで、この章と次の章ではメニューやダイアログボックスの使い方を説明します。

AppWizardが作成するスケルトンには、一般的なメニューがあらかじめ用意されています。プログラマは、このスケルトンメニューをもとに、自分の目的に合った構成のメニューを作ることになります。従来のWindowsプログラミングにおけるメニューまわりのプログラミングと比べ、Visual C++でのメニューの扱いは非常に簡単です。

メニュー処理のプログラミング手順は、プログラムがMDIでもSDIでも大差ありません。ここではSDI形式のプログラムを使って説明をします。

2.1 スケルトンの作成

まずAppWizardを起動して、土台となるスケルトンを作成しましょう。これから作成するプロジェクトは、以下のようなものです。

- プロジェクト名：MenuTest
- アプリケーションタイプ：SDI
- データベースのサポート：しない
- 複合ドキュメントのサポート：しない
- ツールバー／ステータスバー：なし
- 印刷と印刷プレビュー：なし
- そのほか：デフォルトのまま

以上のような条件に従って、AppWizardでオプションを設定してください。設定が済

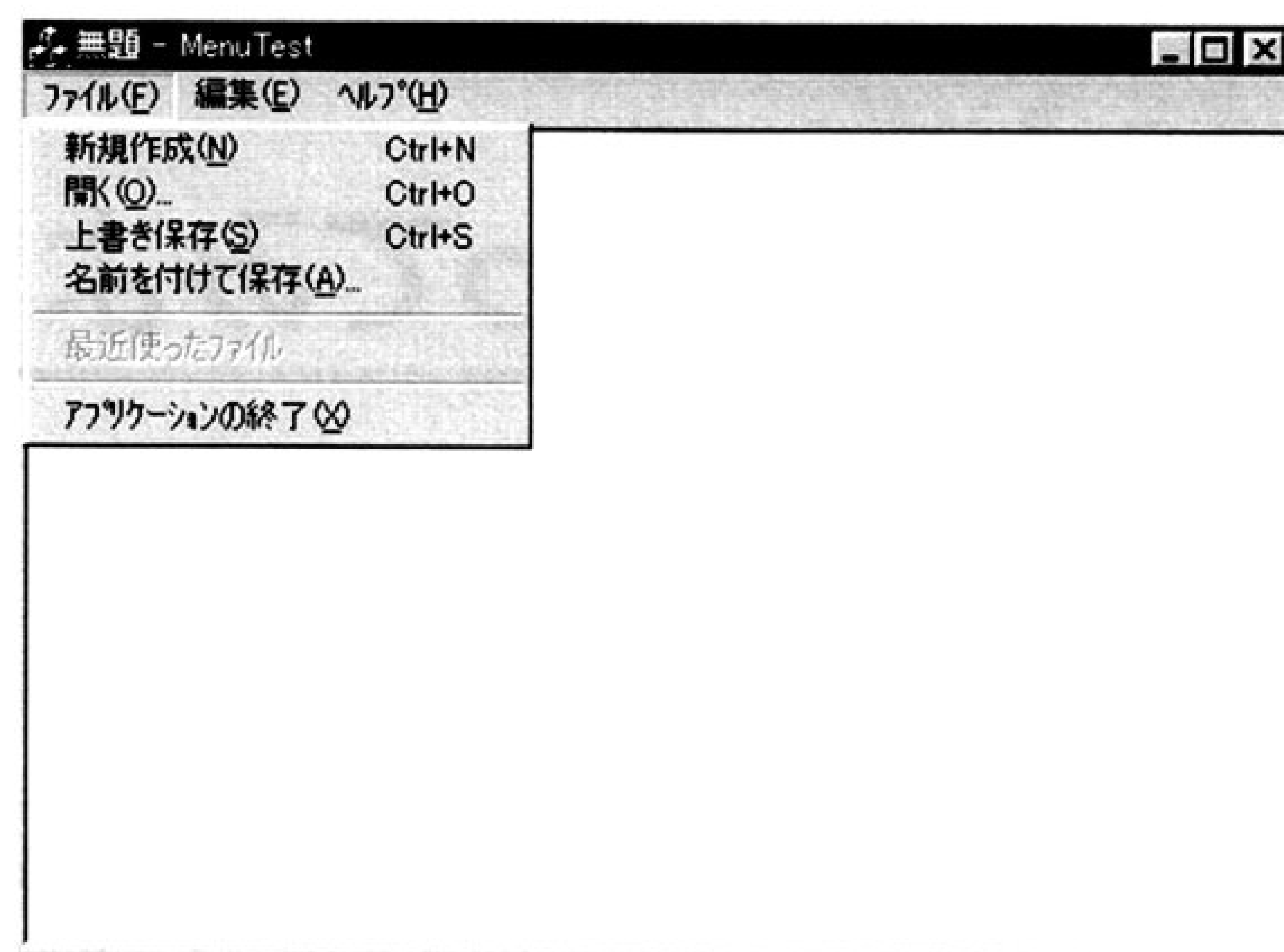


図2-1　デフォルトのメニュー

んだら、例によって＜終了＞、＜OK＞の順にボタンをクリックしてスケルトンを作成してください。スケルトンをコンパイル／実行すると、**図2-1**のような画面が表示されます。

図2-1を見ればわかるように、スケルトンにはデフォルトのメニューが与えられています。メニューバーに表示されるトップメニューには［ファイル］、［編集］、［ヘルプ］の3つがあり、これらのトップメニューをクリックすれば、それぞれプルダウンメニューが表示されます。ただし、これらのメニューはそのままではちゃんと動作しません。というのは、これらのメニュー項目にはまだ対応するコード（そのメニュー項目が選択されると実行されるコード）が割り当てられていないからです。また、これから作成するプログラムには必要がないので削除する項目や、逆に追加しなければいけない項目もあり、この構成のままでは使えません。

そこで、リソースエディタとClassWizardを使用することになります。リソースエディタはメニューを編集（メニュー項目の整理や追加）するのに使い、ClassWizardはリソースエディタで作成したメニューにプログラムコードを割り当てるのに使います。それでは、これから両ツールの使い方を詳しく説明していきましょう。

2.2 メニュー項目の削除

メニューはWindowsのリソースの1つで、リソースエディタでその内容をチェックしたり、修正をしたりすることができます。

メニューを編集するには、Developer Studioのワークスペースウィンドウで［Resource

View］タブをクリックし、［……リソース］というフォルダ（ここでは［MenuTest リソース］）をダブルクリックします。すると、プログラムのリソーススクリプト（ここではMenuTest.rc）の読み込みが始まります。リソーススクリプトは、プロジェクトのリソースの設定を記述したテキストファイルで、「.rc」という拡張子を持っています。

リソーススクリプトの読み込みが終わると、プロジェクトワークスペースウィンドウには、そのプロジェクトで使用されているリソースの種類が一覧表示されます（**図 2-2**）。



図 2-2 リソースの種類を一覧表示したところ

現在の目的はメニューの編集なので、［MenuTest リソース］の下にある［Menu］をダブルクリック、あるいはその左側にある＋記号をクリックします。するとその下に、プロジェクトに含まれるすべてのメニューのリソース ID が表示されます（MenuTest プロジェクトはメニューを 1 つしか持っていないので、「IDR_MAINFRAME」というリソース ID が 1 個表示されるだけである）。第 1 部でも触れましたが、リソース ID とはプロジェクトの随所でリソース指定のために使われる整数値で、Resource.h などのインクルードファイルで名前が付けられています。

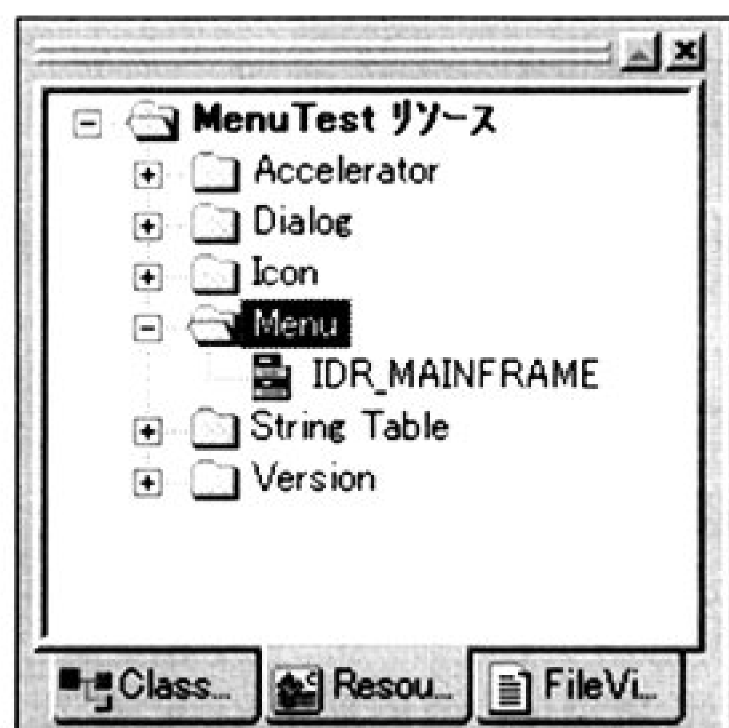


図 2-3 「IDR_MAINFRAME」というリソース ID が表示された

編集したいメニューのリソース ID（ここでは IDR_MAINFRAME）をダブルクリックすると、メニューエディタのウィンドウが開きます（**図 2-4**）。

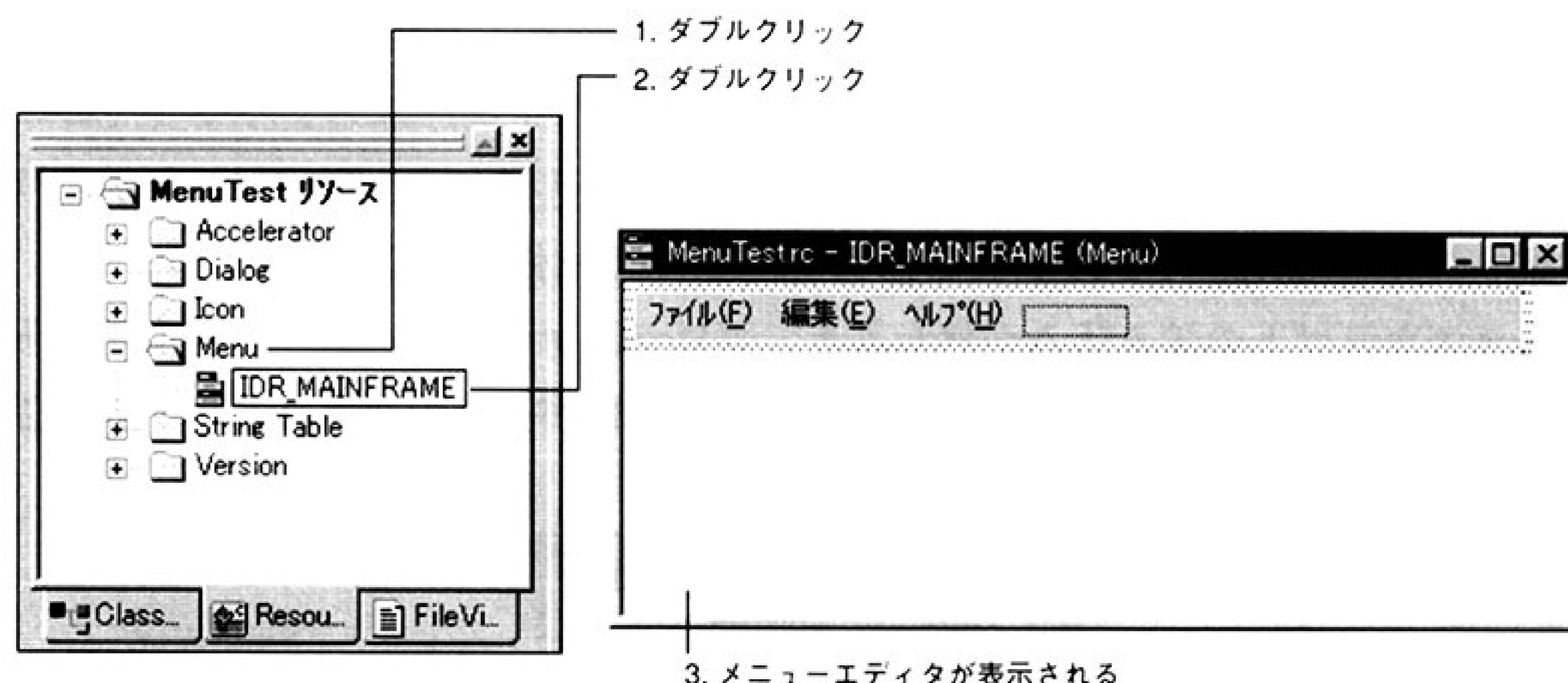


図 2-4　メニューエディタが開いたところ

ウィンドウの上部には選択したメニューがプログラムの実行時とほとんど同じ形で表示されます。トップメニューをクリックすれば、その下のプルダウンメニューも姿を見せます。実際のメニューと違うのは、トップメニューの右端やプルダウンメニューの最下行に空白の四角形（ニューアイテムボックス）が置かれていることですが、これは後述するように、メニュー項目を追加する位置を示します。

また太枠の四角形は、現在注目しているメニュー項目を示すカーソルです。別のメニュー項目をマウスでクリックすると、カーソルはその位置に移動します。Tab や矢印キーを押して、項目ごとにカーソルを順次移動することもできます。

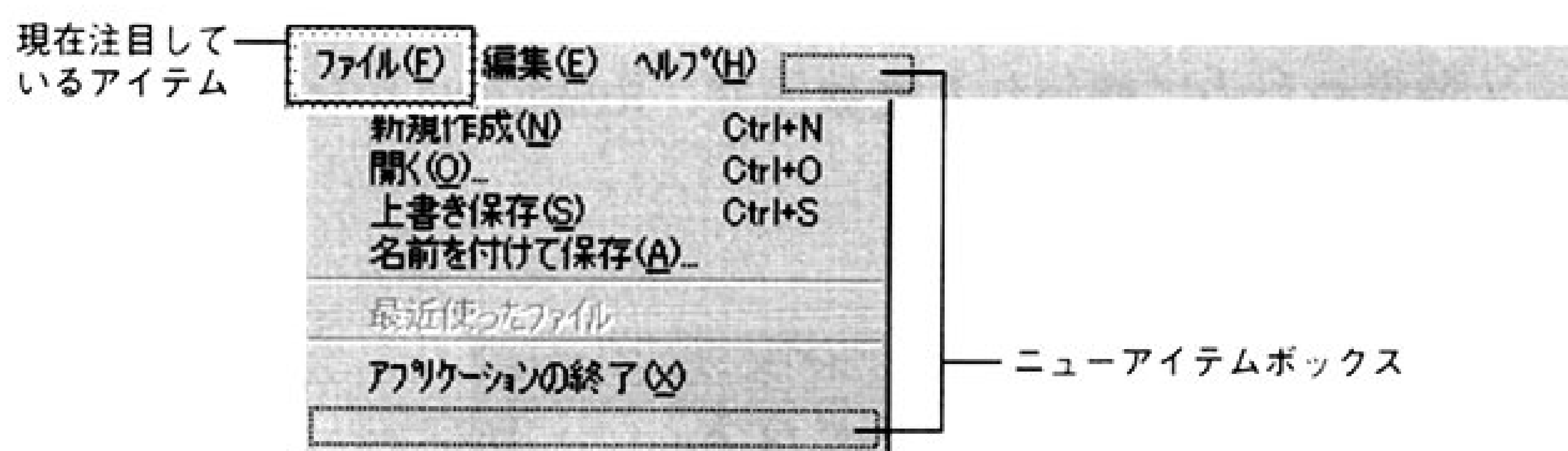


図 2-5　［ファイル］のプルダウンメニューを表示したところ

ところで、この章で作成するプログラムは、ファイルの読み書きはいっさいしません。よって、ファイルアクセスに関するメニュー項目は必要ありません。これらは邪魔なだけなので、削除してしまいましょう。

まず［ファイル］メニューの［新規作成］にカーソルを合わせ、Delete を押します。これで［新規作成］のメニュー項目が削除されるはずです（**図 2-6**）。以下同様に、［開く］、［上書き保存］……と削除していきます。項目を区切るセパレータ（横線）も、やはりその上にカーソルを合わせて Delete を押せば削除できます。**図 2-6** に示すようにして、［アプリケーションの終了］だけ残し、あとはすべて削除してください。

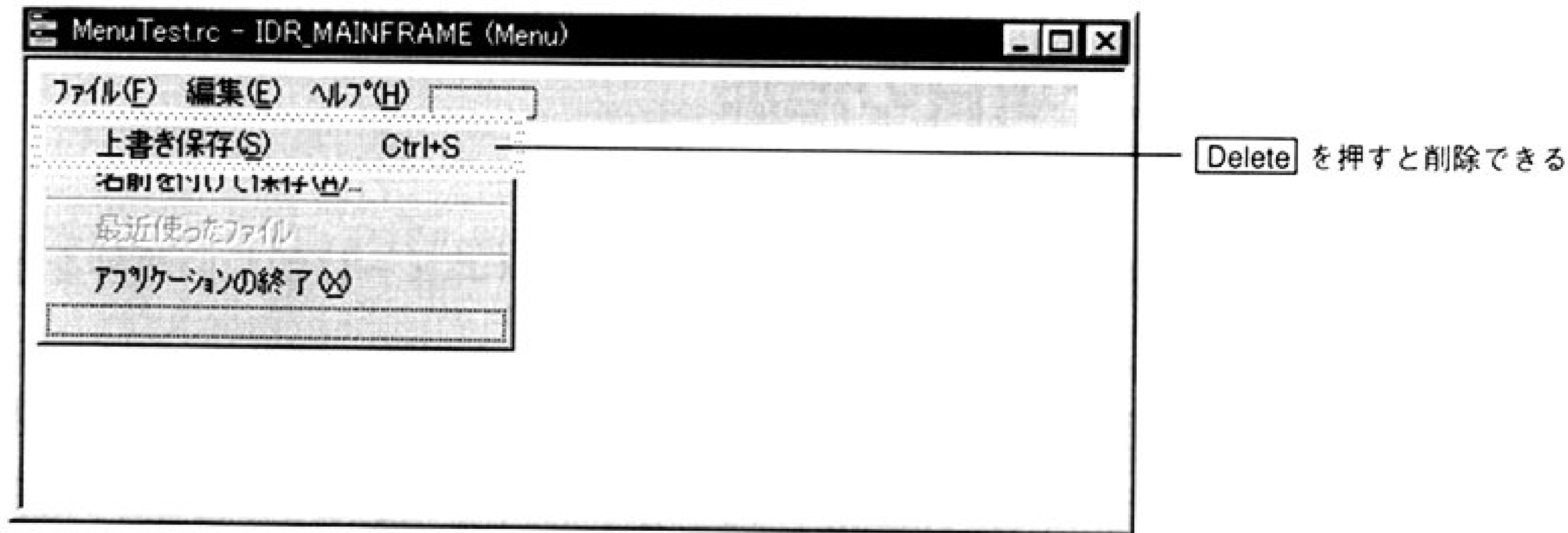


図 2-6　ファイル操作用のいくつかのメニュー項目を削除したところ

お隣の[編集]メニューの中には、今回必要とする項目は1つもありません。よって、こちらは根こそぎ削除してしまいます。トップメニューの[編集]にカーソルを合わせて Delete を押すと、**図 2-7** のようなダイアログボックスが表示されるので、＜OK＞ボタンをクリックしてください。これで[編集]メニューはその下のプルダウンメニューを含め完全に削除されます。

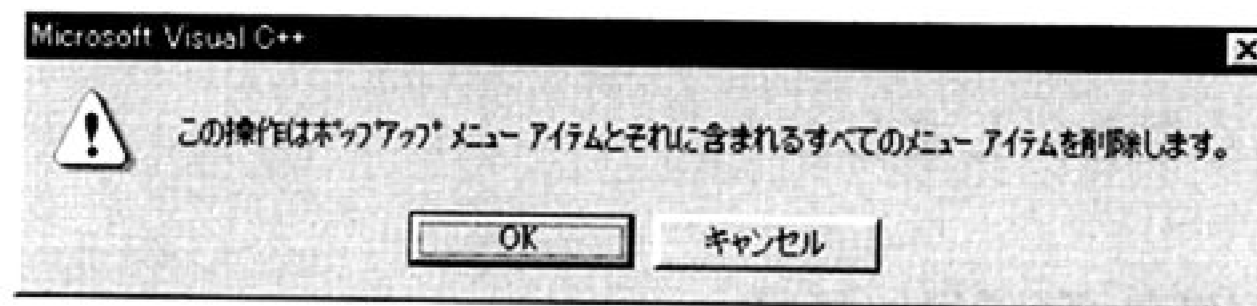


図 2-7　トップメニュー削除の確認ダイアログボックス

以上で不要なメニュー項目はすべて取り除かれました。プログラムを再コンパイルして実行してみてください。表示されるメニューは**図 2-8** のように変更されているはずです。

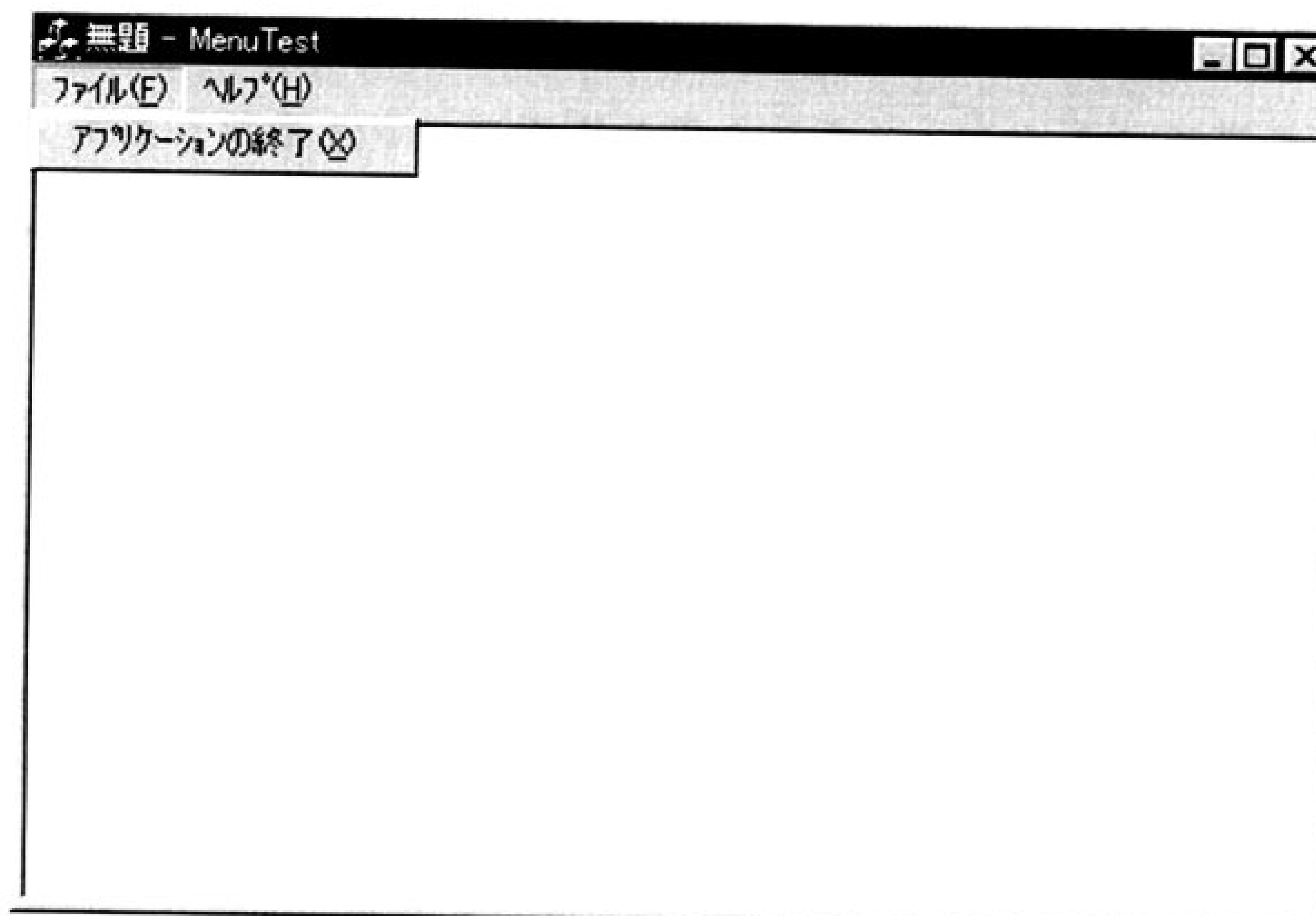


図 2-8　メニュー変更後のプログラムの実行画面

2.3 メニュー項目の追加

不要な項目が整理され、すっきりしたところで、今度はメニューに新しい項目を追加してみましょう。

メニューを使った操作の対象として、**リスト 2-2** のような図形表示プログラムを用意しました。CMenuTestView::OnDraw 関数は、myCircle 関数と myLine 関数を呼び出して顔を描きます。そこでまず、MenuTestView.cpp の**リスト 2-1** に示す部分を**リスト 2-2** のように修正してください。

これらの関数は、円と直線を組み合わせて顔を描くだけのものですが、口の形、右目の向き、左目の向き、眉の上下、眉の太さという 5 つのパラメータを持ち、それらを個別に変更することができます。各パラメータは**リスト 2-2** の先頭に示すグローバル変数で管理しています。本来これらのデータはドキュメントクラスを利用して管理をするべきものですが、ここでは話を簡単にするためにグローバル変数として管理することにしました。

そこで、メニューを利用してこれらの変数の値を増減させ、いろいろな表情を作ってみようというわけです。いっぺんに完成させようと思うとたいへんですから、簡単なところから順々に仕上げていくことにしましょう。

リスト 2-1　修正前のリスト（MenuTestView.cpp）

```
/////////////////////////////////////////
// CMenuTestView クラスの描画

void CMenuTestView::OnDraw(CDC* pDC)
{
  CMenuTestDoc* pDoc = GetDocument();
  ASSERT_VALID(pDoc);

  // TODO: この場所にネイティブデータ用の描画コードを追加します。
}
```

リスト 2-2　顔を描く関数群（MenuTestView.cpp）

```
/////////////////////////////////////////
// CMenuTestView クラスの描画

int  MouthPos = -10;        // 口の上げ下げ
int  RtEyePos = 10;         // 右目の左右の位置
int  LtEyePos = -10;        // 左目の左右の位置
int  BrowPos   = 0;         // 眉の上げ下げ
int  BrowWidth  = 3;        // 眉の太さ
```

```
void myCircle(CDC* pDC, int x, int y, int r, int brush)
{
  pDC->SelectStockObject(brush);
  pDC->Ellipse(x - r, y - r, x + r, y + r); // (x, y) を中心に半径 r の円を描く
}

void myLine(CDC* pDC, int x, int y, int dx, int dy, int width)
{
  CPen  Pen(PS_SOLID, width, RGB(0,0,0));    // 指定した太さ (width) の
  CPen* pOldPen = pDC->SelectObject(&Pen);   // 黒いペンを選択
  pDC->MoveTo(x - dx, y - dy);               // (x-dx, y-dy) から
  pDC->LineTo(x + dx, y + dy);               // (x+dx, y+dx) へ直線を引く
  pDC->SelectObject(pOldPen);
}

void CMenuTestView::OnDraw(CDC* pDC)
{
  CRect rect;
  int   x, y;

  GetClientRect(&rect);
  x = rect.right / 2;   // ビューウィンドウの中心を得る
  y = rect.bottom / 2;

  myCircle(pDC, x, y, 140, LTGRAY_BRUSH);                      // 顔
  myCircle(pDC, x - 50, y - 20, 30, WHITE_BRUSH);              // 右の白目
  myCircle(pDC, x + 50, y - 20, 30, WHITE_BRUSH);              // 左の白目
  myCircle(pDC, x - 50 + RtEyePos, y - 20, 20, BLACK_BRUSH);  // 右の瞳
  myCircle(pDC, x + 50 + LtEyePos, y - 20, 20, BLACK_BRUSH);  // 左の瞳
  myLine(pDC, x - 50, y - 80,  30, BrowPos, BrowWidth);        // 右の眉毛
  myLine(pDC, x + 50, y - 80, -30, BrowPos, BrowWidth);        // 左の眉毛
  myLine(pDC, x - 40, y + 70,  40, MouthPos, 3);               // 口の右半分
  myLine(pDC, x + 40, y + 70, -40, MouthPos, 3);               // 口の左半分
}
```

最初は「にっこりした口(MouthPos＝10)」と「むすっとした口(MouthPos＝-10)」のどちらかを選ぶようなメニューを考えます。目標は**図 2-9** のようなメニューです。

それでは、Developer Studio からメニューエディタのウィンドウを開いてください。

メニューバーの右端にある空白の四角形は**ニューアイテムボックス**といって、新しいメニュー項目は、この位置に作成することになります。とりあえず、ニューアイテムボックスをマウスでダブルクリックしてください。すると、**図 2-10** のようなプロパティウィンドウが表示されます。

プロパティウィンドウの[キャプション]ボックスでは、メニューのキャプション文字列を指定します。ここでは「口元の形」と入力しましょう。するとメニューエディタのメニューにも、同じ文字列が表示されます。

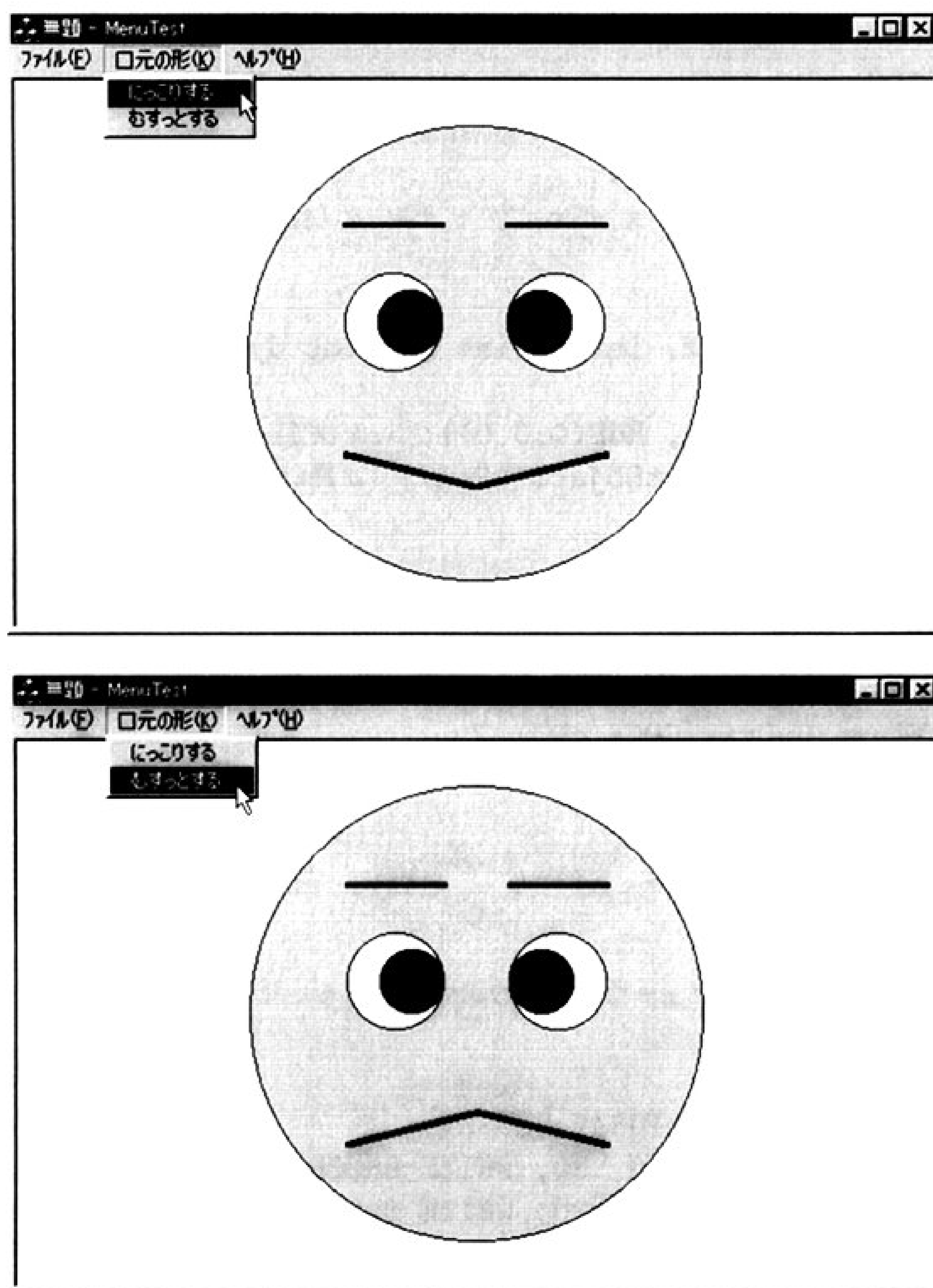


図 2-9 ［口元の形］メニューとそれぞれの表示図形



図 2-10 ニューアイテムボックスとプロパティウィンドウ

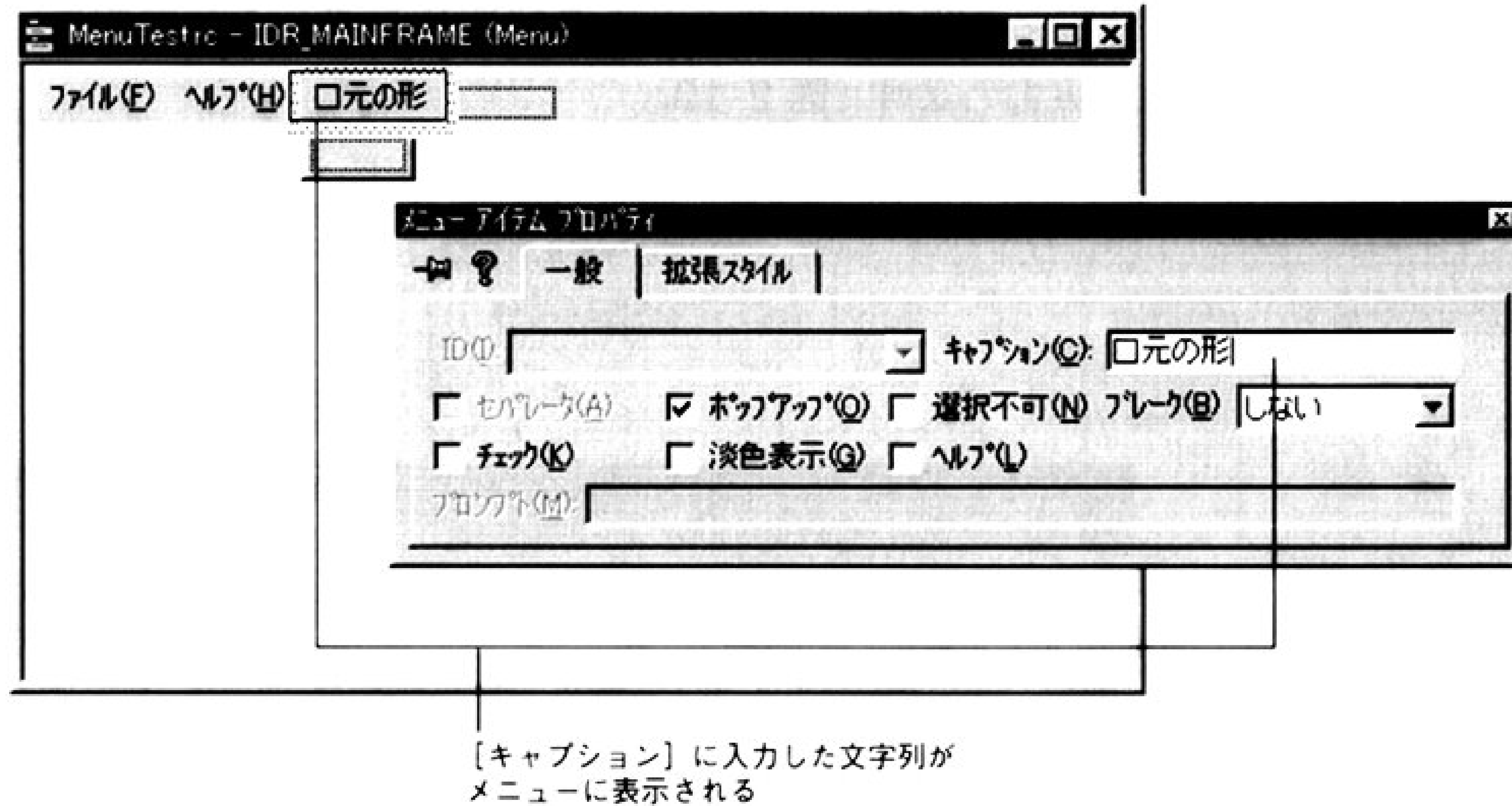


図 2-11 「口元の形」というキャプションを入力したところ

キャプションにはアクセスキーの指定を含めることができます。**アクセスキー**とは、メニューを選択するのに利用できる Alt +任意の 1 文字のことで、アクセスキーを押すことは、そのメニューをマウスでクリックするのと同じ意味を持ちます。

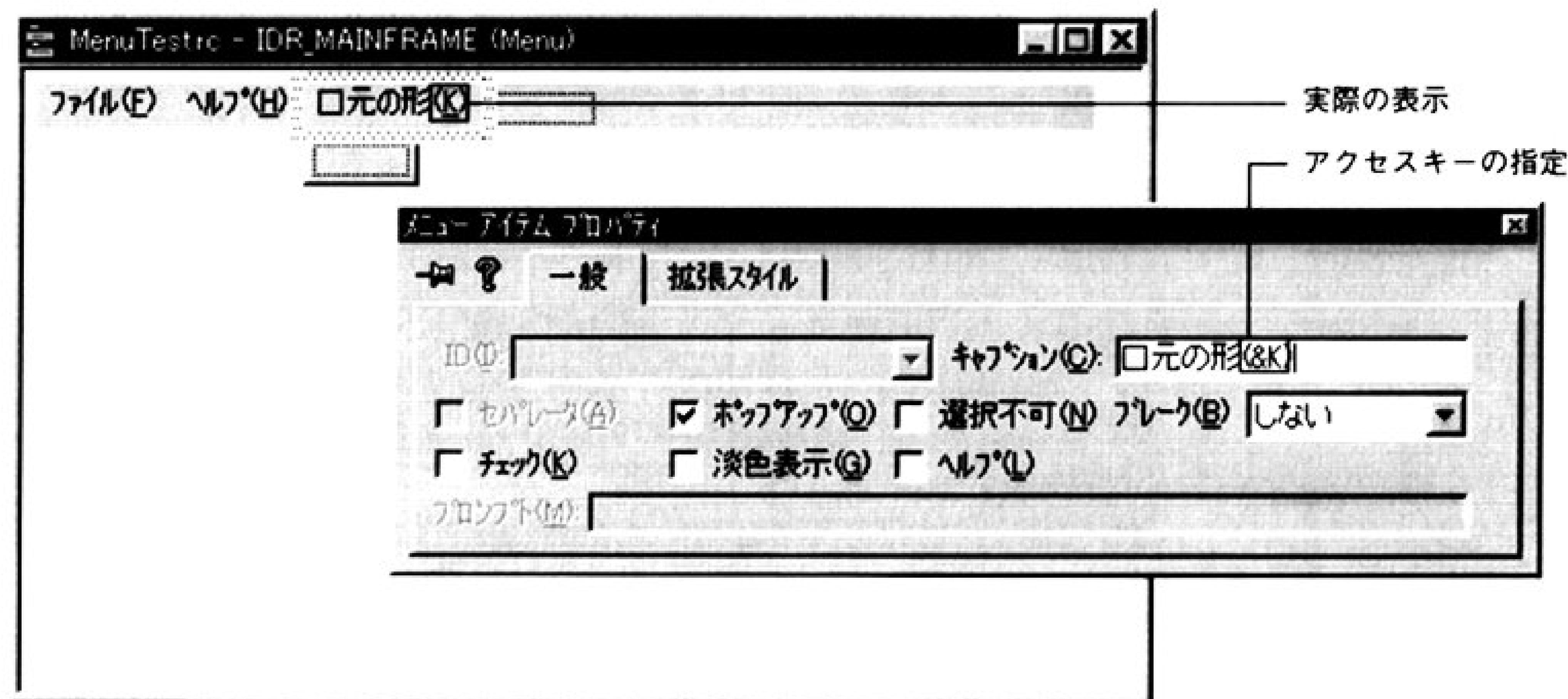


図 2-12 アクセスキーを含むキャプション指定と、その実際のメニュー表示

アクセスキーを指定するには、「&[アクセスキー]」という文字を含めてキャプションを入力します。「口元の形(&K)」というキャプションを指定すると、このメニューのアクセスキーは Alt + K になります。またこのようにアクセスキー指定をキャプションに含めると、実際のメニューでは「&K」ではなく下線が付いた「K」が表示されます。

トップレベルのメニュー項目を作成すると、そのすぐ下にニューアイテムボックスが1つ生じますから、ここにプルダウンメニュー項目を作成します。ここでは[にっこりする]、[むすっとする]という2つのメニュー項目を作りましょう。作成方法はトップメニューと

同様で、ニューアイテムボックスをまずダブルクリックしてプロパティボックスを表示したら、そこでキャプションを入力します（**図 2-13**）。

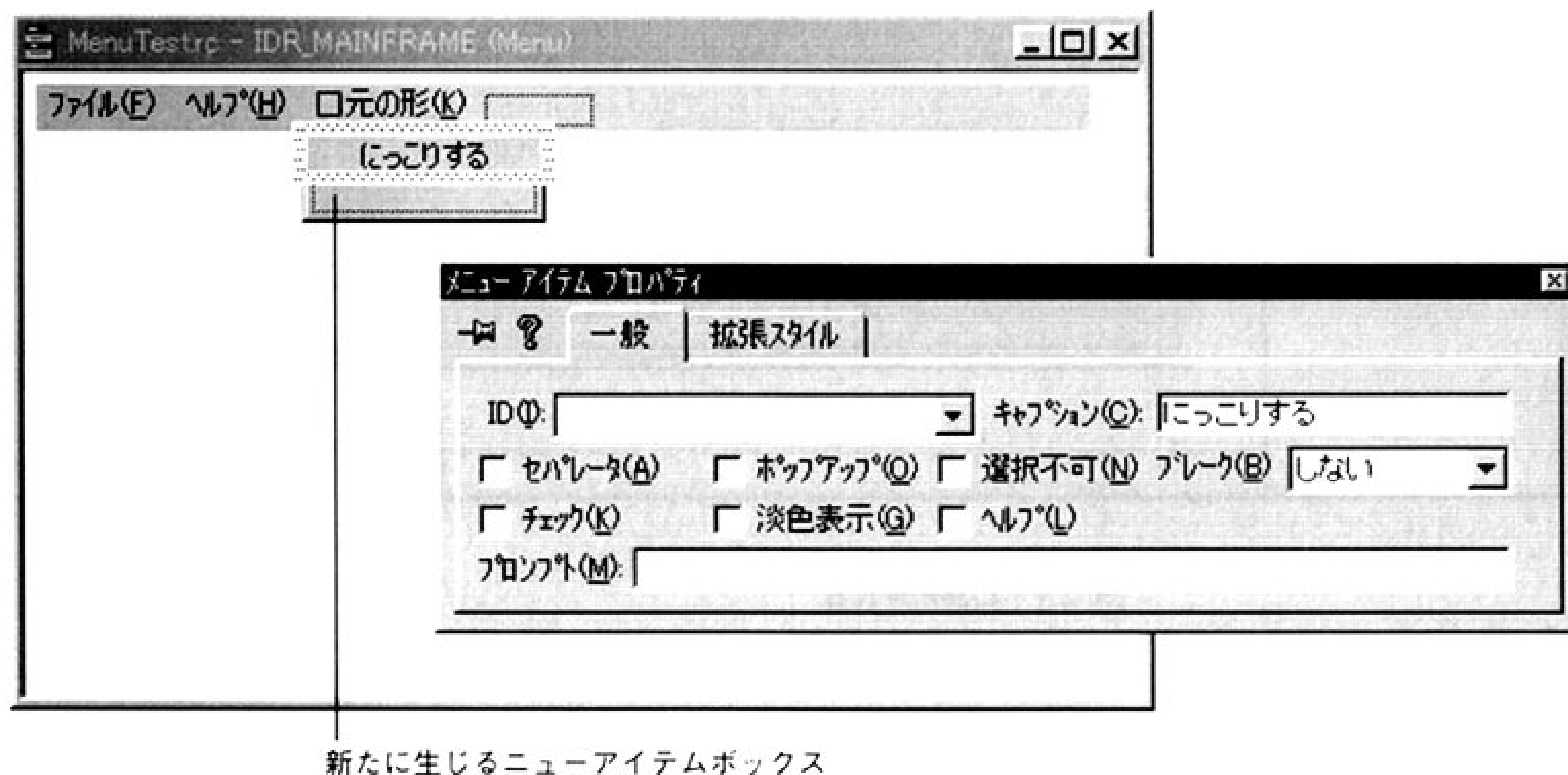


図 2-13　トップレベルのメニューの下に生じたニューアイテムボックス

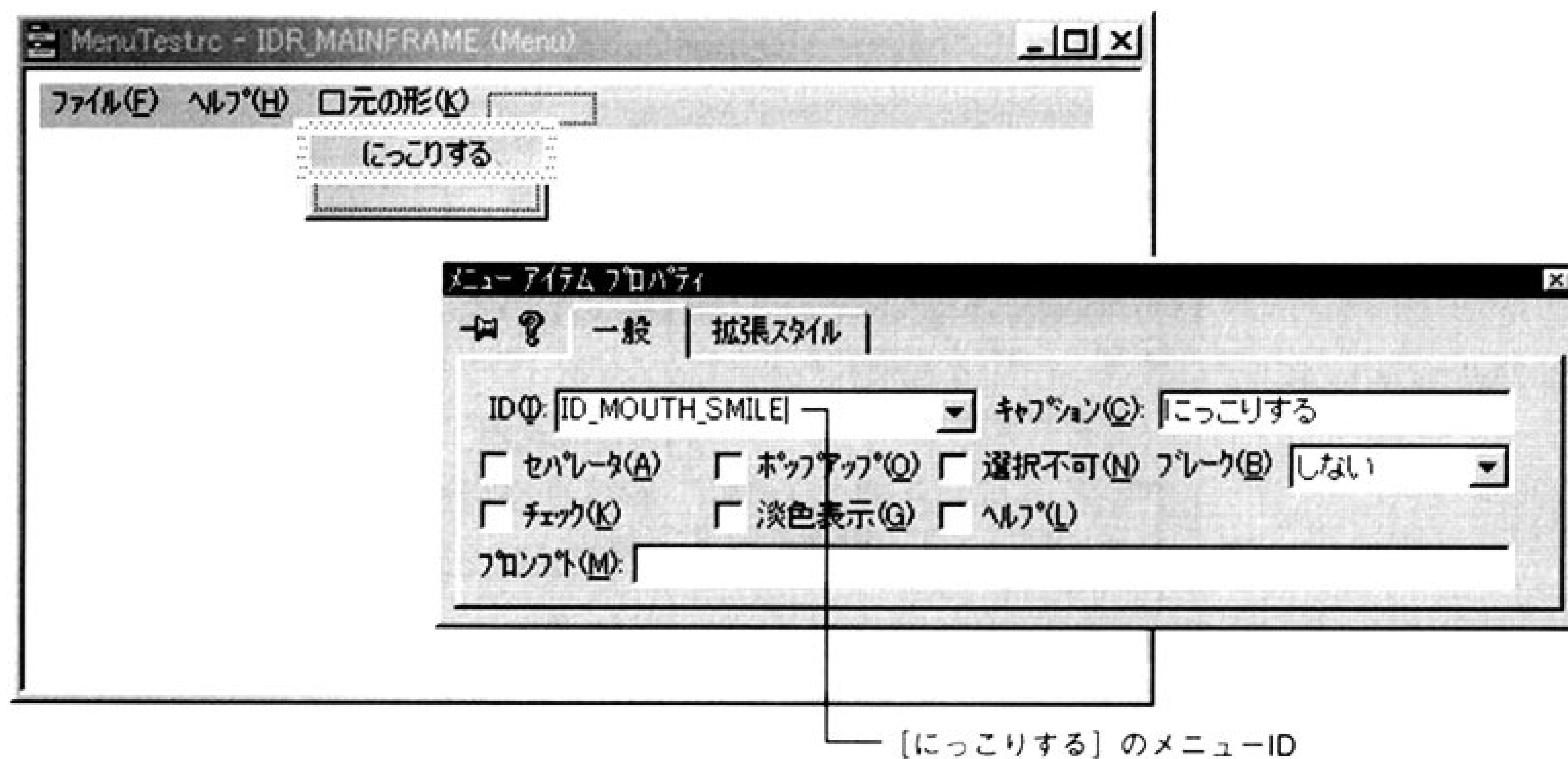


図 2-14　キャプションのほかにメニュー ID も指定する

これらのメニュー項目では、さきほどのトップメニューと異なり、キャプションのほかにオブジェクト ID も指定する必要があります。メニュー項目をクリックすると、**COMMAND メッセージ**というメッセージが発生し、そのメニュー項目のオブジェクト ID を伴ってアプリケーションに送られます。アプリケーションはこのオブジェクト ID を見て、どのメニュー項目が選択されたのかを判断して、実行する関数を決定します。

オブジェクト ID の名前はあとで ClassWizard でも使われるので、メニュー項目が選択されたときに行う動作がわかるようにしておくことをお勧めします。ここでは［にっこりする］

メニューに「ID_MOUTH_SMILE」、[むすっとする]メニューに「ID_MOUTH_ANGRY」という名前を付けておきましょう。リソースエディタは指定されたID名に適当な整数値を割り当て、それをResource.hに記録します。

最後に（実際にはいつでもよいが）メニューの並び順を整えます。各メニュー項目はマウスでドラッグすることにより、位置を自由に変えることができます。Windowsアプリケーションでは通常［ファイル］メニューは一番左、［ヘルプ］メニューは一番右という慣習があるので、MenuTestプログラムでもその例にならうことにします。

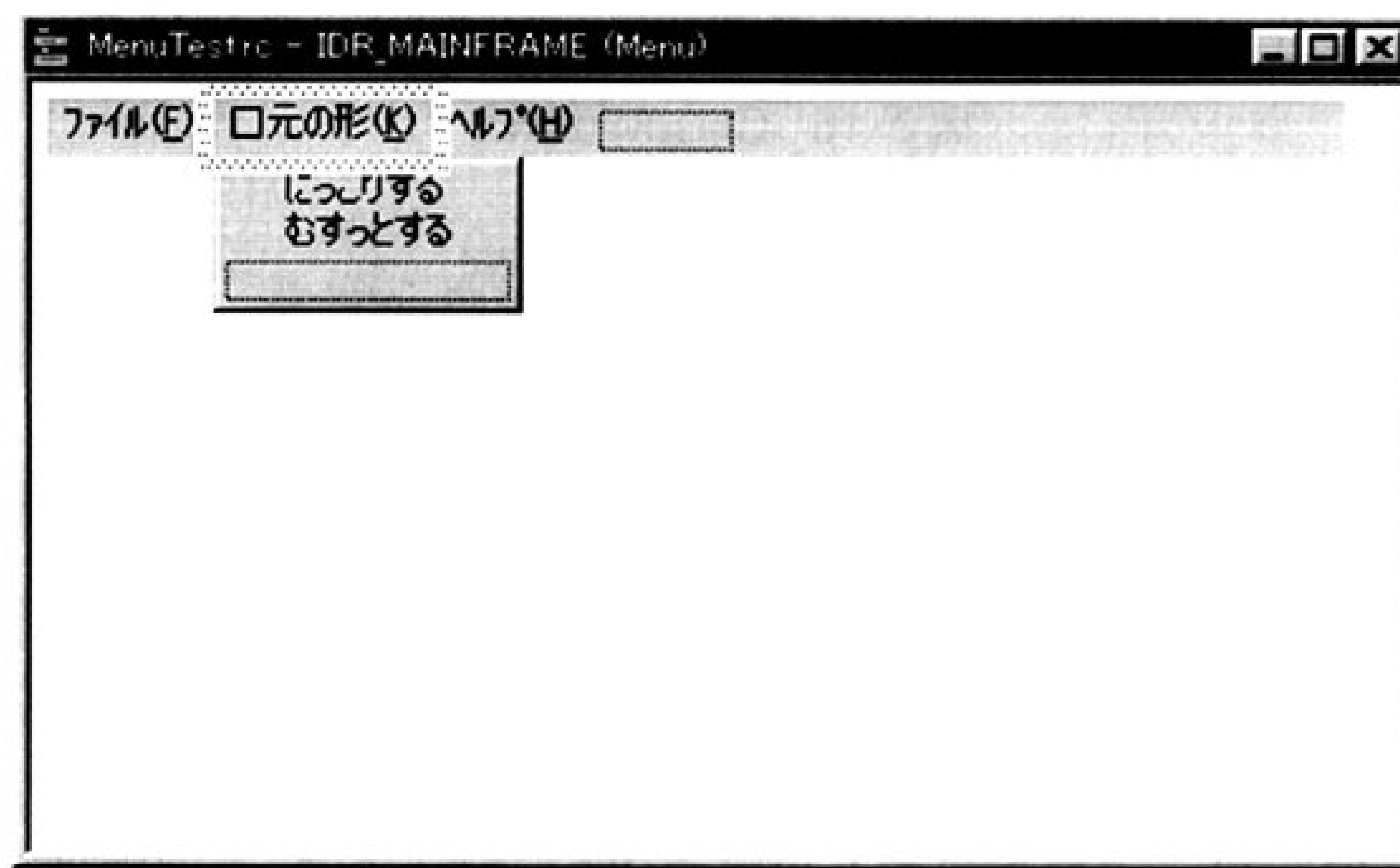


図2-15　メニューの順番を整えたところ

リソースエディタを使用してのメニューを追加する作業は以上で終わりです。ここまでが、工程の約半分。あとの仕事はClassWizardの受け持ちになります。

2.4 メッセージハンドラの記述

ここからはメニュー項目を選択したときに発生するメッセージを処理するための**メッセージハンドラ**をClassWizardを利用して作成することが主な作業となってきます。

ClassWizardを起動するには、メニューから［表示］－［ClassWizard］を選択するか、あるいは Ctrl ＋ W を押します。ClassWizardが起動すると、**図2-16**のようなダイアログボックスが表示されます。

このダイアログボックスで［にっこりする］メニューを選択するとほほ笑むようなメッセージハンドラを作ります。まずClassWizardの［クラス名］コンボボックスで、［にっこりする］メニューをクリックしたときに発生するCOMMANDメッセージの受け手となるクラスを指定します。このメニューが選択されるとクライアント領域に描かれた顔の表情が変化する（再描画される）ということは、**このメッセージはビュークラス（CMenuTestView**

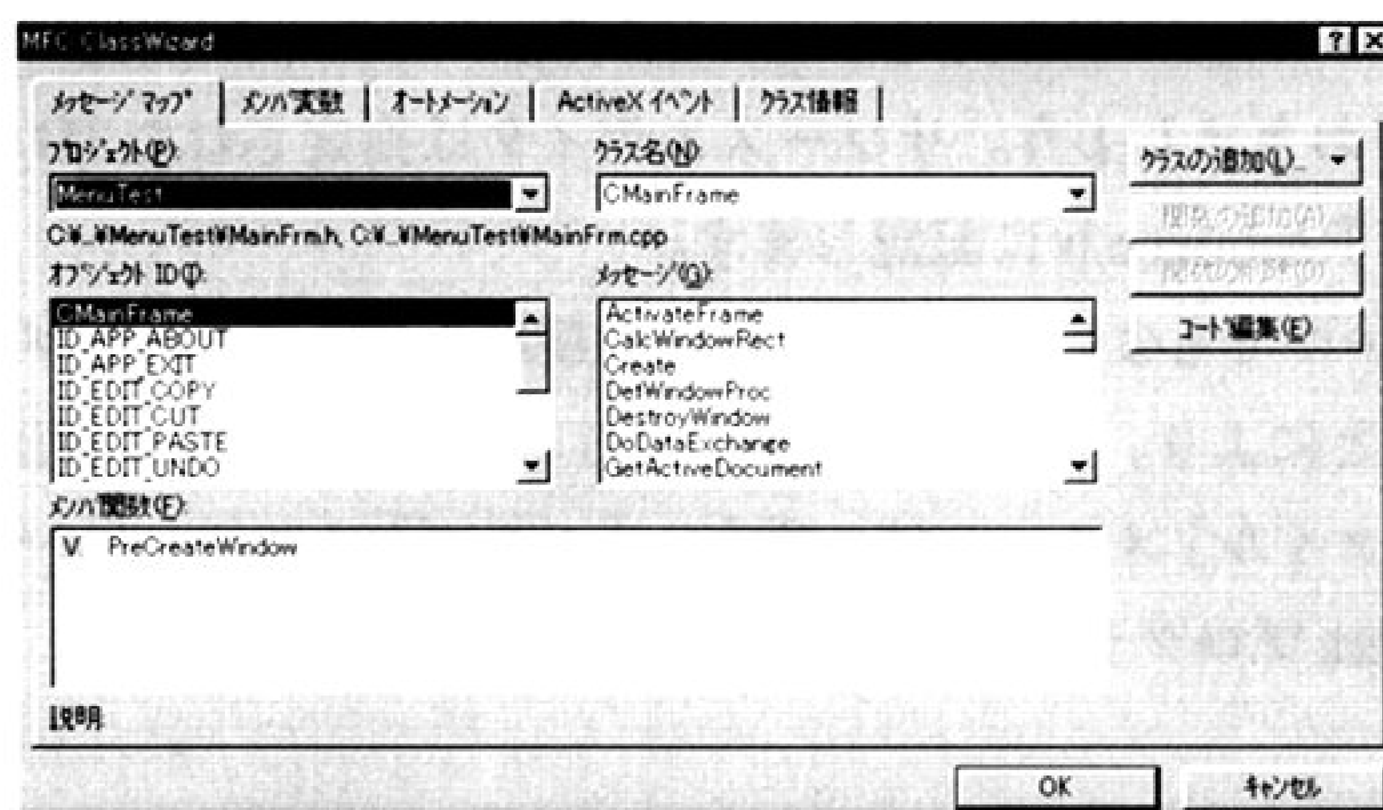


図 2-16　ClassWizard

クラス）のオブジェクトが処理をする必要があるということです。よって、ここではクラス名には「CMenuTestView」を指定するのがよいでしょう。

クラスを指定すると［オブジェクト ID］リストボックスに、そのクラスのオブジェクトにメッセージを送ることができるオブジェクト ID の一覧が表示されます。ここでは［にっこりする］メニューがメッセージの送り手なので、「ID_MOUTH_SMILE」を選択します。なお、リストの先頭には受け手のクラス名（CMenuTestView）と同じものが表示されていますが、これはオブジェクト ID が必要ないメッセージ（たとえば WM_PAINT）とコードを結び付ける場合に選択します。

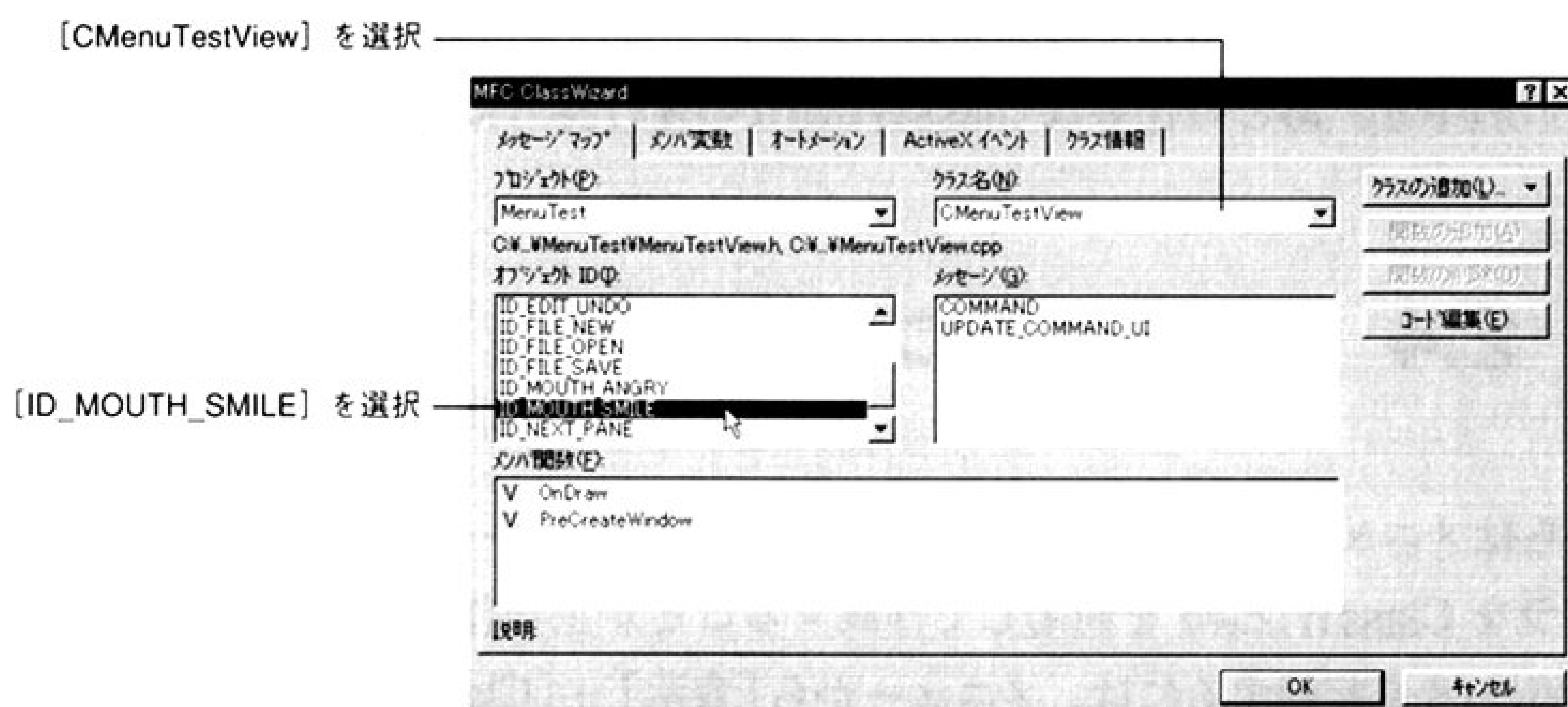


図 2-17　［クラス名］と［オブジェクト ID］を指定した

Windows のメッセージには、メッセージの名前だけ知れば意味がわかるものと、送り手が誰なのか調べないと対処のしようがないものの 2 種類があります。たとえば、「画面を描き直せ」という意味の WM_PAINT メッセージは前者の例で、このメッセージを受け取ったウィンドウは、それがどこから送られてきたものだろうと、かまわず画面を描き直せばよいわけです。一方、メニュー項目が発するメッセージは COMMAND メッセージと

呼ばれ、後者に属します。ウィンドウは、多くのメニュー項目から同じCOMMANDメッセージを受け取りますが、これだけでは、どのオブジェクトがどんな目的でこのメッセージを発したのかはわかりません。そのため、COMMANDメッセージには、その送り手を特定するためのオブジェクトIDが付いてくるのです。そして、ウィンドウは、このIDをもとにCOMMANDメッセージにどう反応すればよいか判断するのです。

メッセージの受け手となるクラス名と、送り手のオブジェクトIDが決まると、[メッセージ]の欄にオブジェクトが送り出すことができるメッセージの一覧が表示されます。メニューが送り手の場合は、送り出すことができるメッセージは**表2-1**に示す2種類しかありません。

メニューが送出できるメッセージ	メッセージの意味
COMMAND	メニューが選択された
UPDATE_COMMAND_UI	インターフェイス更新を通知する

表2-1 メニューが送ることができるメッセージ

メニューが選択されたことをビューオブジェクトに伝えるのはCOMMANDメッセージですから、「COMMAND」を選択します。なおUPDATE_COMMAND_UIメッセージについては後述します。次にメッセージを処理する関数(メッセージハンドラ)の定義を行うため、<関数の追加>ボタンをクリックします。すると関数名を指定する**図2-18**のようなダイアログボックスが開きます。

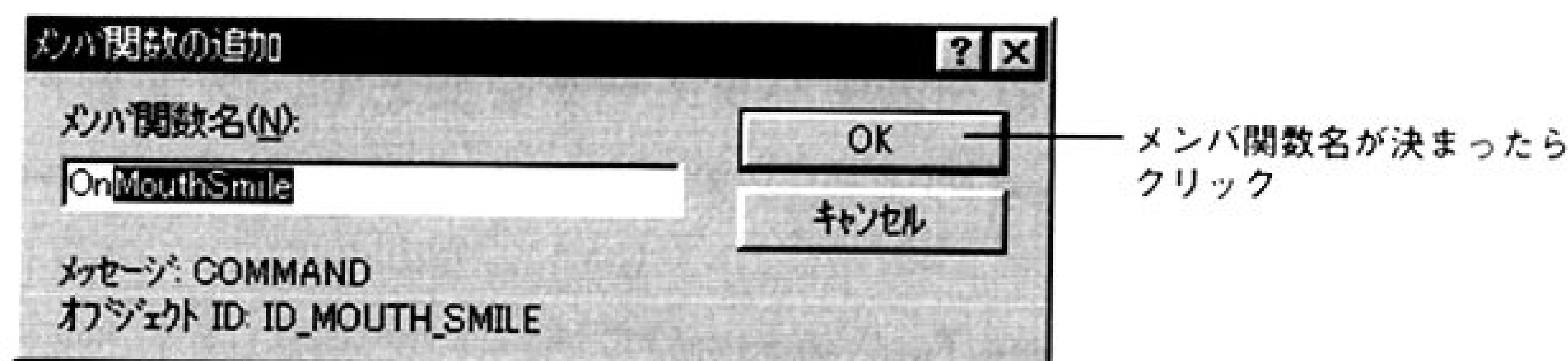


図2-18 [メンバ関数の追加]ダイアログボックス

関数名は「On」で始まり、適当な文字列が続きます。ClassWizardが示してくれる名前(ここではOnMouthSmile)を、そのまま利用してもいいでしょう。<OK>ボタンをクリックすると、指定した名前のメッセージハンドラが[メンバ関数]の欄に追加され、同時にメッセージハンドラのスケルトンが作成され、ファイルに書き込まれます。

[メンバ関数]のリストから、今作成したメッセージハンドラ(OnMouthSmile)をマウスで選択し、<コード編集>ボタンをクリックすると、エディタが起動してメッセージハンドラの位置にカーソルが移動し、**リスト2-3**のような関数が表示されます。

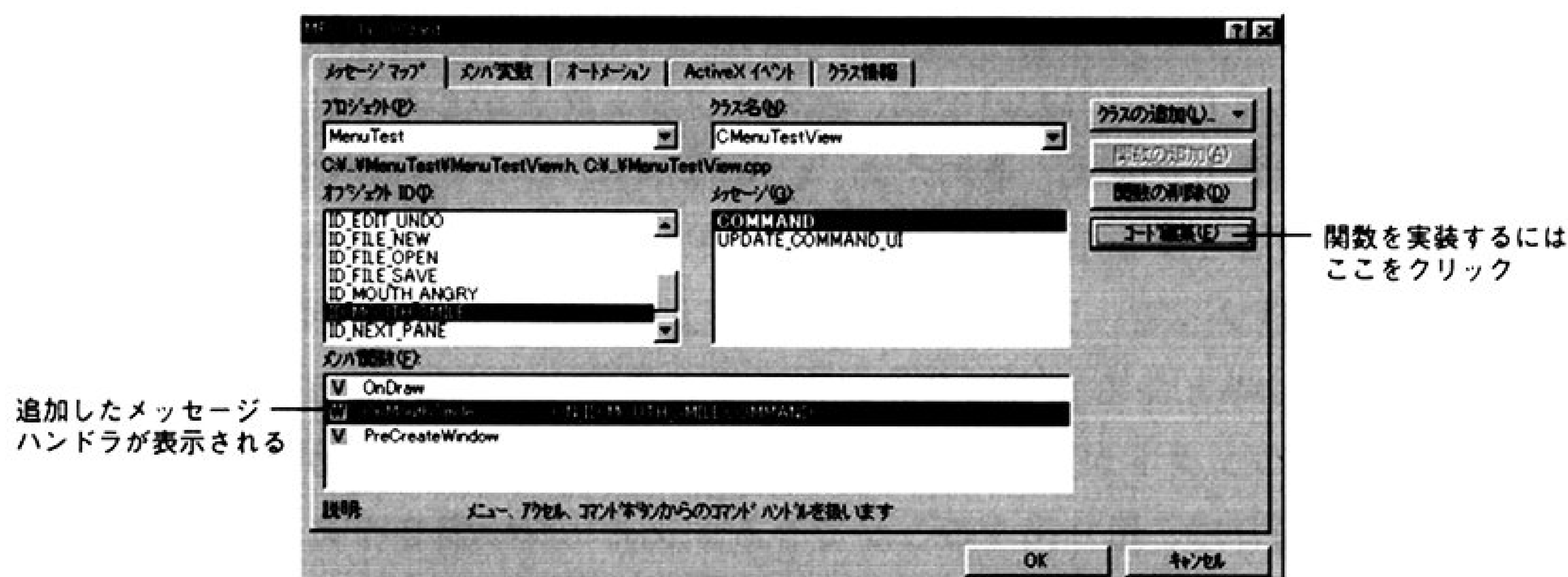


図2-19 ［メンバ関数］にメニューのメッセージハンドラが追加された

リスト2-3 ［にっこり］メニューのメッセージハンドラのスケルトン（MenuTestView.cpp）

```
void CMenuTestView::OnMouthSmile()
{
  // TODO: この位置にコマンド ハンドラ用のコードを追加してください

}
```

COMMANDメッセージを受けとったビューオブジェクトは、付属するオブジェクトIDを見て、それが［にっこりする］メニューから送られてきたことを知り、このOnMouthSmileメッセージハンドラを実行します。

ここからあとは、プログラマであるあなた自身の受け持ちです。といっても、もうほとんど仕事は残っていません。口の位置を表すパラメータを変えて、クライアント領域を更新するだけ。たった2行、**リスト2-4**のようにプログラムを書き直せばオーケーです。

リスト2-4 修正後のメッセージハンドラ（MenuTestView.cpp）

```
void CMenuTestView::OnMouthSmile()  //修正後のメッセージハンドラ
{
  MouthPos = 10;                   //口の端を基準線から10ドット上げる
  InvalidateRect(NULL, FALSE);  //ウィンドウを描き直す
}
```

InvalidateRect関数は、CWndクラスのメンバ関数で、ウィンドウの表示内容を更新するようシステムに要求します。なお上記の引数は「全ウィンドウ更新、背景消去なし」を意味します（詳しくはオンラインマニュアルの「MFCリファレンス」を参照のこと）。ここでは操作対象を指定していませんが、これは「this->InvalidateRect（NULL, FALSE）」とみなされ、メッセージハンドラを実行しているビューオブジェクトに対してInvalidateRect

関数が実行されます。これによって、ビューオブジェクトのクライアント領域が更新され、CMenuTestView::OnDraw 関数が実行されます。

というわけで、ようやく［にっこりする］メニューのメッセージハンドラが完成しました。同様の手順で、［むすっとする］メニューのメッセージハンドラ CMenuTestView::OnMouthAngry 関数も作成してみてください。今度は**表 2-2** に示すデータを指定すればよいわけです。**図 2-20** に、WizardBar を利用したメッセージハンドラの作成方法を示します。

クラス名	CMenuTestView
オブジェクト ID	ID_MOUTH_ANGRY
メッセージ	COMMAND
関数名	OnMouthAngry

表 2-2　ID_MOUTH_ANGRY が発生する COMMAND メッセージのメッセージハンドラ

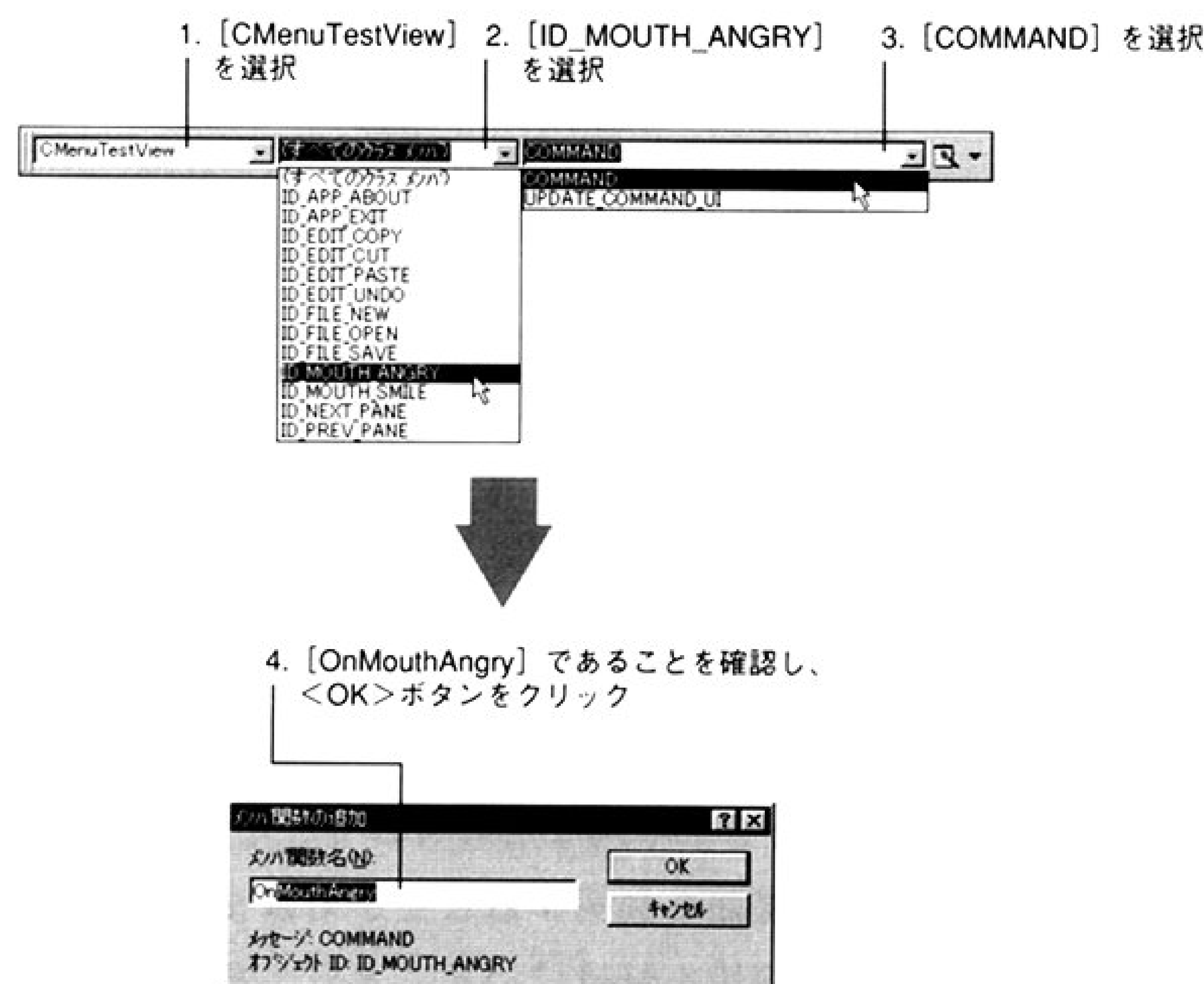


図 2-20　WizardBar によるメッセージハンドラの作成

CMenuTestView::OnMouthAngry 関数は**リスト 2-5** のように記述してください。

リスト 2-5　［むすっとする］メニューのメッセージハンドラ（MenuTestView.cpp）

```
void CMenuTestView::OnMouthAngry()
{
  MouthPos = -10;            // 口の端を基準線から 10 ドット下げる
  InvalidateRect(NULL, FALSE);  // ウィンドウを描き直す
}
```

2つのメッセージハンドラの作成が終わったら、プロジェクトをコンパイルして実行してみましょう。［口元の形］メニューを選択するとプルダウンメニューが表示されること、そして［にっこりする］や［むすっとする］を選択すると、それに応じて口元の形が変わることを確認してください(**図 2-21**)。

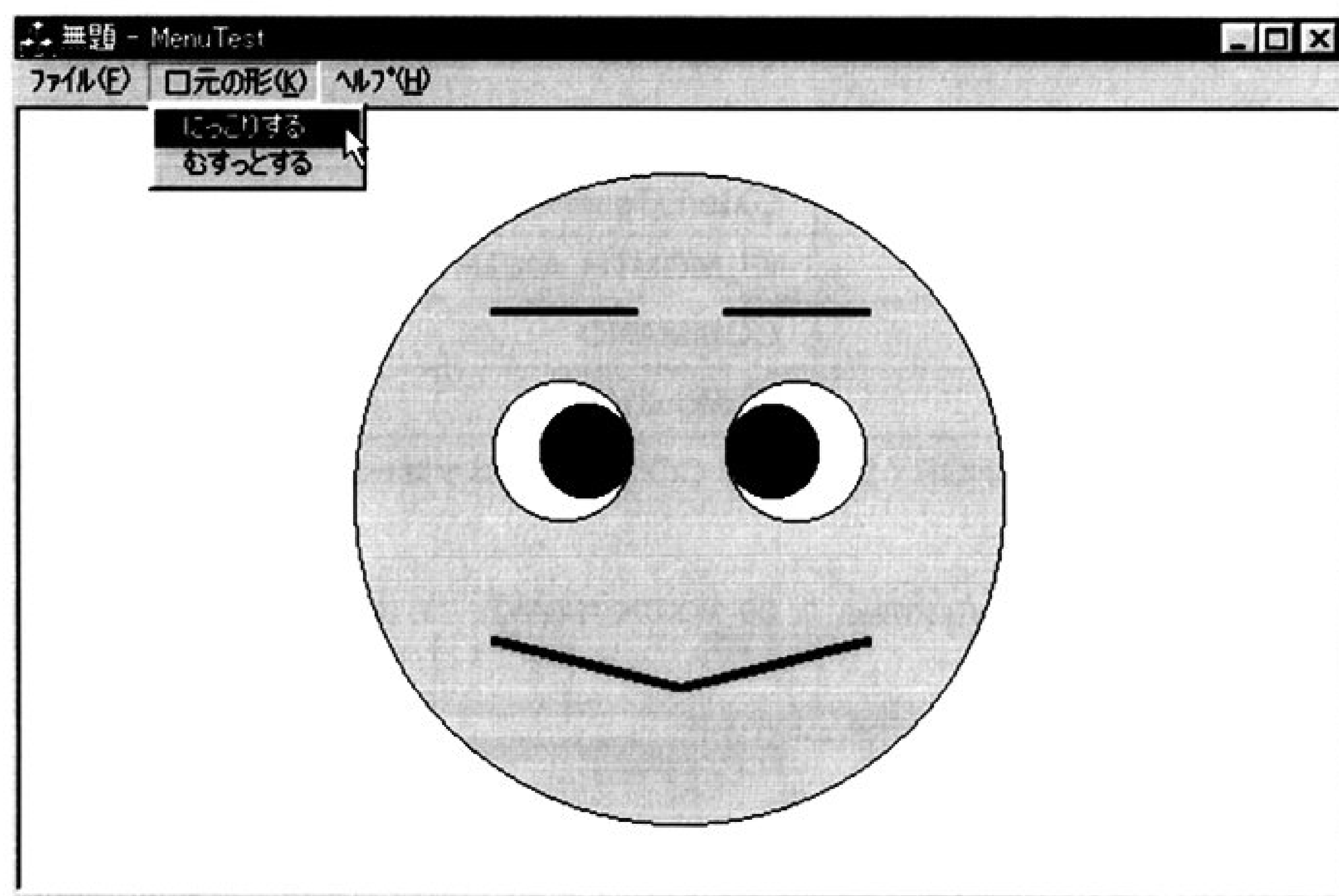


図 2-21　にっこりしたところ

2.5 メニューの付加機能

前節で作成したメニューは、もっとも基本的な形のものでしたが、Windowsのメニューでは、以下に示すようないろいろな機能を利用することができます。この節では、これらの機能を使いながら顔にいろいろな表情を付けていくことにします。

- チェックマークを表示する
- 状況によって動作を止める(メニューの有効化／無効化)
- ショートカットキーを定義する

● チェックマークを追加する

まずチェックマークの利用例として、MenuTestプロジェクトに**図 2-22**に示すメニューを追加してみましょう。

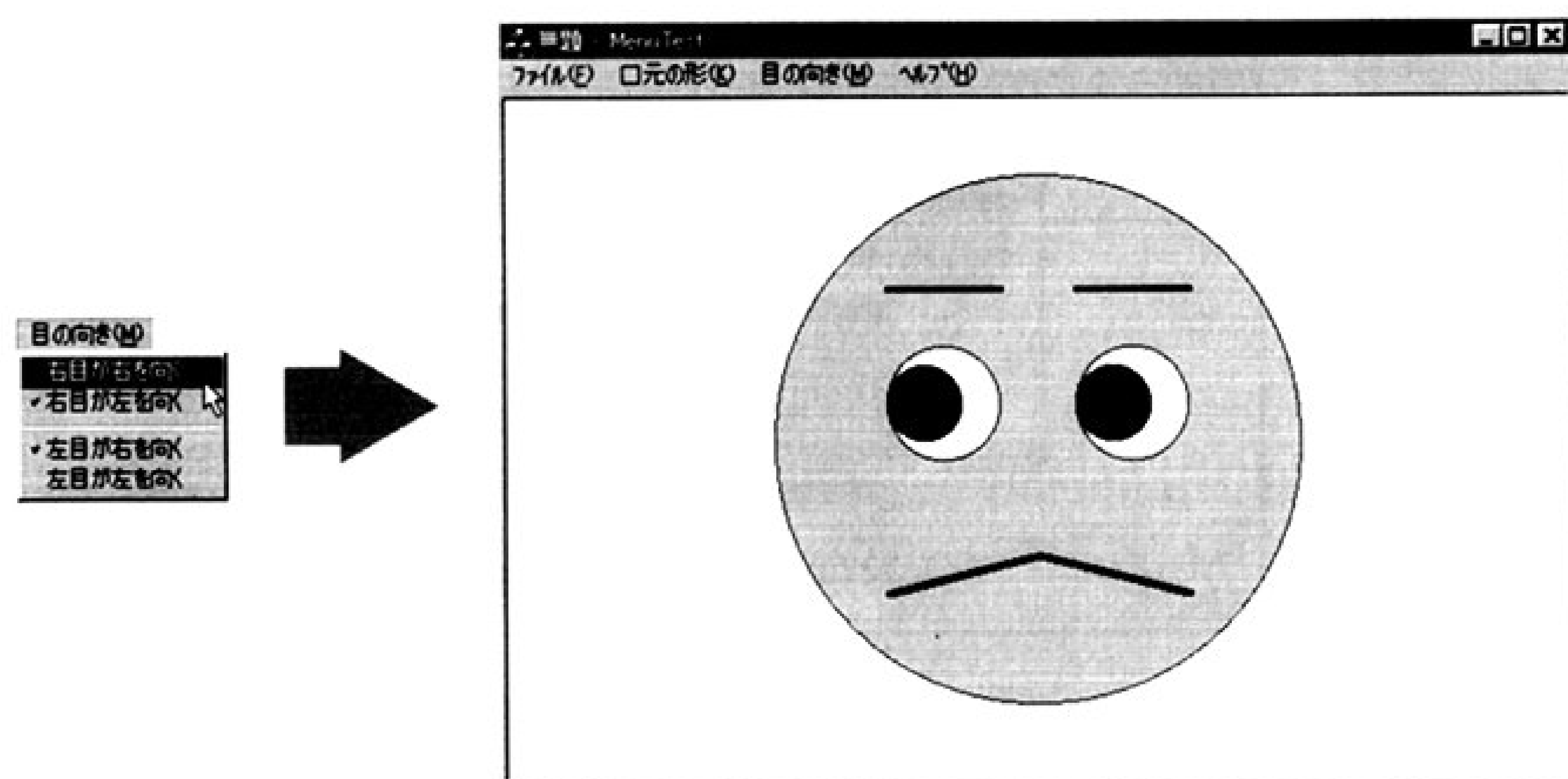


図 2-22 ［目の向き］メニューとその表示図形

メニューの作成

このメニューは CMenuTestView::OnDraw 関数が表示する目玉の位置を変えます。プルダウンメニューの 4 つの項目によって、左右の目それぞれについて向きを指定します。そして現在の目の向きと一致するメニュー項目には、チェックマークを表示します。

それではメニューエディタを起動して、各メニュー項目に**表 2-3** のキャプションとメニュー ID を設定してください。

キャプション	メニュー ID
目の向き(&M)	なし(トップメニュー)
右目が右を向く	ID_RIGHT_EYE_RIGHT
右目が左を向く	ID_RIGHT_EYE_LEFT
左目が右を向く	ID_LEFT_EYE_RIGHT
左目が左を向く	ID_LEFT_EYE_LEFT

表 2-3 メニュー項目とその ID

プルダウンメニューの 2 つ目の項目と 3 つ目の項目の間には、メニューを見やすくするため、セパレータ（区切り線）を入れましょう。手順は非常に簡単です。まず、ニューアイテムボックスをセパレータを挿入する位置に移動して、ニューアイテムボックスが選択されている状態で、プロパティボックスの［セパレータ］をチェックします。すると、ニューアイテムボックスの位置にセパレータが追加されます（**図 2-23**）。

メニューが完成したら、ClassWizard を起動してください。そして前節と同様に、4 つのメニューの COMMAND メッセージに対するメッセージハンドラを作成します。メッセージハンドラ名と ClassWizard で指定する各要素の対応は**表 2-4** に示します。ですが、ClassWizard をいちいち起動して、オブジェクト ID を指定して、……というのが面倒くさいという人は、WizardBar を使用してもよいでしょう。

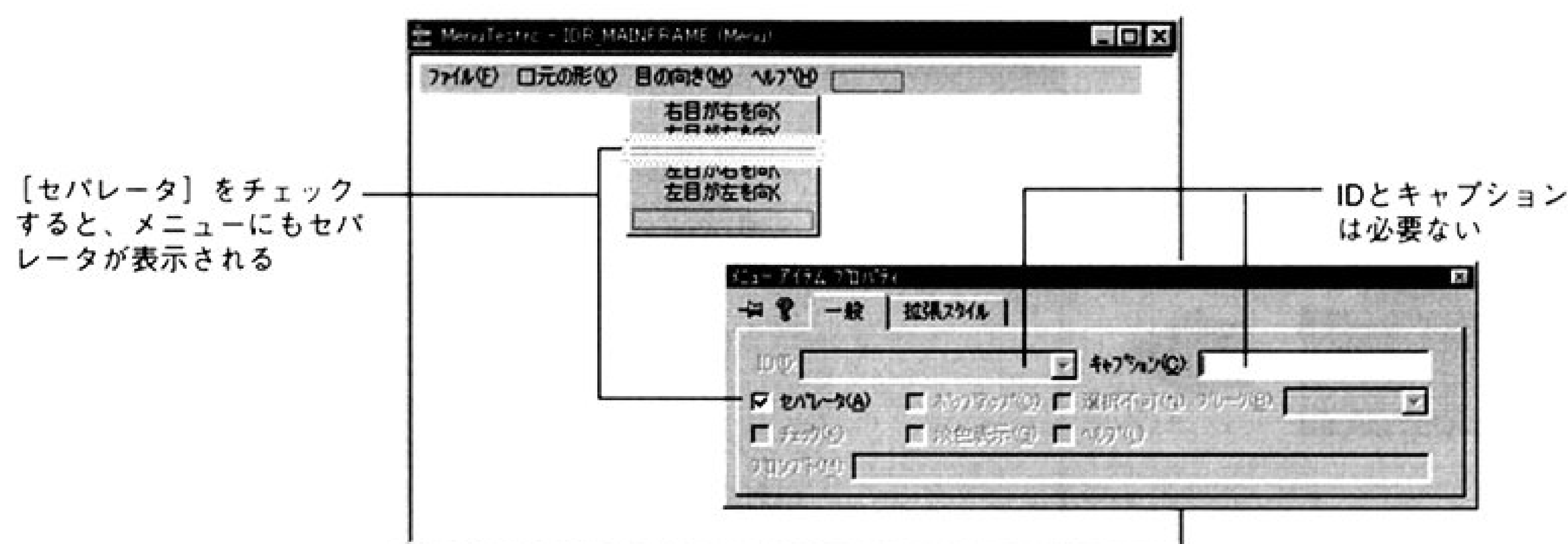


図 2-23 プロパティボックスの [セパレータ] をチェックしたところ

クラス名	オブジェクト ID	メッセージ	メッセージハンドラ
CMenuTestView	ID_RIGHT_EYE_RIGHT	COMMAND	OnRightEyeRight
CMenuTestView	ID_RIGHT_EYE_LEFT	COMMAND	OnRightEyeLeft
CMenuTestView	ID_LEFT_EYE_RIGHT	COMMAND	OnLeftEyeRight
CMenuTestView	ID_LEFT_EYE_LEFT	COMMAND	OnLeftEyeLeft

表 2-4 目の向きを変えるメッセージハンドラ

リスト 2-6 目の向きを変更するメッセージハンドラ (MenuTestView.cpp)

```
void CMenuTestView::OnRightEyeRight()
{
  RtEyePos = -10;
  InvalidateRect(NULL, FALSE);
}

void CMenuTestView::OnRightEyeLeft()
{
  RtEyePos = 10;
  InvalidateRect(NULL, FALSE);
}

void CMenuTestView::OnLeftEyeRight()
{
  LtEyePos = -10;
  InvalidateRect(NULL, FALSE);
}

void CMenuTestView::OnLeftEyeLeft()
{
  LtEyePos = 10;
  InvalidateRect(NULL, FALSE);
}
```

チェックマークを付けるには?

これで、メニューの基本動作部分が完成しましたが、本題はここからです。このメニューにチェックマークを表示させるにはどうすればよいでしょうか?

その答は、メニューが送り出すことのできるもう1つのメッセージ、**UPDATE_COMMAND_UI** を利用することです。UPDATE_COMMAND_UI メッセージは、メニューやツールバーなどのユーザーインターフェイスが何らかの変化をおこす直前に生じます。

たとえばプルダウンメニューが表示されるとき、それぞれのプルダウンメニュー項目は、画面に姿を現す直前に UPDATE_COMMAND_UI メッセージを発生します。また、このメッセージが送られるとき、それぞれのメニュー項目は自分自身へのポインタも同時に送り出します。UPDATE_COMMAND_UI メッセージのメッセージハンドラでは、渡されたポインタを介して送り手にアクセスし、その状態を変更することができます。メニューにチェックマークを付ける場合も、このしくみを利用します。

実際に、[右目が右を向く]メニューにチェックマークを表示させてみましょう。まず**表 2-5** に示す指定に従って、UPDATE_COMMAND_UI メッセージハンドラのスケルトンを作ります。UPDATE_COMMAND_UI の場合は、メッセージハンドラの関数名は「OnUpdate」で始まることに注意してください。

クラス名	CMenuTestView
オブジェクト ID	ID_RIGHT_EYE_RIGHT
メッセージ	UPDATE_COMMAND_UI
関数名	OnUpdateRightEyeRight

表 2-5 チェックマークを表示するメッセージハンドラの指定

作成されるメッセージハンドラのスケルトンは**リスト 2-7** のようなものになります。COMMAND メッセージのメッセージハンドラと異なり、こちらは pCmdUI という引数を持っていることに注意してください。

リスト 2-7 UPDATE_COMMAND_UI メッセージハンドラのスケルトン(MenuTestView.cpp)

```
void CMenuTestView::OnUpdateRightEyeRight(CCmdUI* pCmdUI)
{
  // TODO: この位置にcommand update UIハンドラ用のコードを追加してください

}
```

引数 pCmdUI は、メッセージの送り手(ここでは[右目が右を向く]メニュー)を示すもので、CCmdUI クラスのオブジェクトへのポインタです。CCmdUI クラスは、メニューとツールバーなどの種類が異なるオブジェクトを統一的に扱えるようにするためのクラス

です。

pCmdUI が指すメニューにチェックマークを付けるには、CCmdUI::SetCheck 関数を使います。この関数は引数を 1 つ持ち、その値が 1 ならメニューを「チェック ON」の状態とし、0 なら「チェック OFF」とします*1。メニューが表示されるときには、この設定状態によって、チェックマークを付けるかどうかが決まります。

［右目が右を向く］メニューにチェックマークを付けるメッセージハンドラを**リスト 2-8** に示します。ほかのメニューにチェックマークを付ける場合も、同様なメッセージハンドラを作ればよいわけです。メッセージハンドラの指定は**表 2-6** に示します。

クラス名	オブジェクト名	メッセージ	メッセージハンドラ
CMenuTestView	ID_RIGHT_EYE_RIGHT	UPDATE_COMMAND_UI	OnUpdateRightEyeRight
CMenuTestView	ID_RIGHT_EYE_LEFT	UPDATE_COMMAND_UI	OnUpdateRightEyeLeft
CMenuTestView	ID_LEFT_EYE_RIGHT	UPDATE_COMMAND_UI	OnUpdateLeftEyeRight
CMenuTestView	ID_LEFT_EYE_LEFT	UPDATE_COMMAND_UI	OnUpdateLeftEyeLeft

表 2-6 チェックマークを付けるためのメッセージハンドラ

リスト 2-8 UPDATE_COMMAND_UI メッセージのハンドラ（MenuTestView.cpp）

```
void CMenuTestView::OnUpdateRightEyeRight(CCmdUI* pCmdUI)
{
  pCmdUI->SetCheck((RtEyePos < 0) ? 1 : 0); // RtEyePos<0 ならチェック
}

void CMenuTestView::OnUpdateRightEyeLeft(CCmdUI* pCmdUI)
{
  pCmdUI->SetCheck((RtEyePos>0) ? 1 : 0);    // RtEyePos>0 ならチェック
}

void CMenuTestView::OnUpdateLeftEyeRight(CCmdUI* pCmdUI)
{
  pCmdUI->SetCheck((LtEyePos<0) ? 1 : 0);    // LtEyePos<0 ならチェック
}

void CMenuTestView::OnUpdateLeftEyeLeft(CCmdUI* pCmdUI)
{
  pCmdUI->SetCheck((LtEyePos>0) ? 1 : 0);    // LtEyePos>0 ならチェック
}
```

*1 状態が 2 ということもあり得る（3 ステートボタンの場合）

メニューの無効化とショートカットキーおよびサブメニューの追加

さてここで、MenuTest にもう1つプルダウンメニューを追加することにしましょう。今度はメニューの有効化／無効化の制御と、ショートカットキー定義の実例です。

このメニューは CMenuTestView::OnDraw 関数が表示する眉毛の設定を変更するもので、構成は**図 2-24** のようになります。

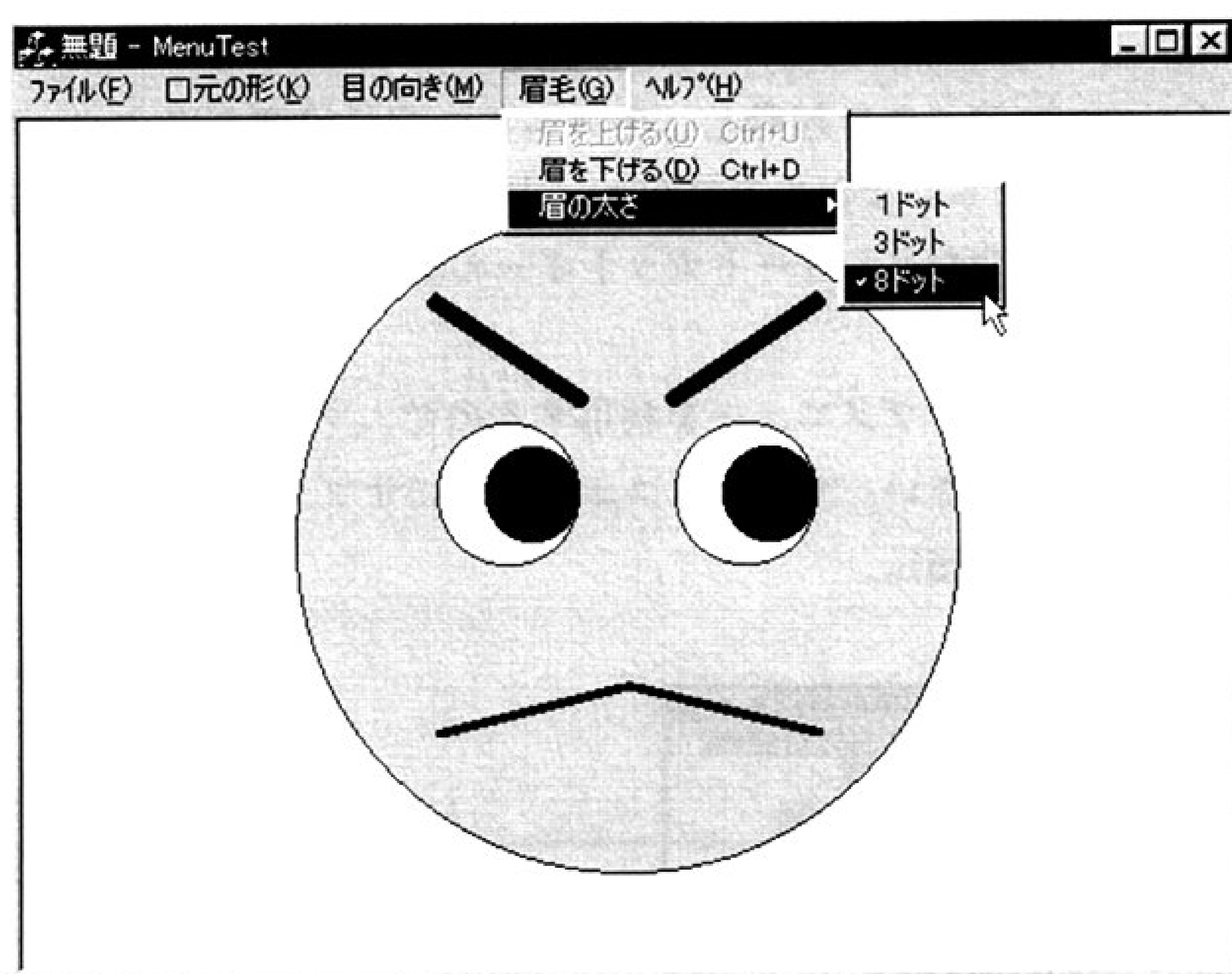


図 2-24 ［眉毛］メニュー

［眉を上げる］メニューを選択すると、選択するごとに眉の端が少しずつ上がっていき、［眉を下げる］メニューを選択すると、その逆に眉の端が少しずつ下がっていきます。どちらのメニューも、眉の位置が決められた限度に達すると、それ以上選択できないように無効化されます。

さらに［眉を上げる］メニューには Ctrl ＋ U 、［眉を下げる］メニューには Ctrl ＋ D というショートカットキーを定義しましょう。ショートカットキーは、アクセスキーと似て否なるもので、これを使えば、メニュー項目が画面に表示されていなくてもそれを直接選択することができます。アクセスキーの場合はいちいちメニューを開かなければ利用できず、たとえば［眉毛］－［眉を上げる］を選択するには、 Alt ＋ G で［眉毛］のプルダウンメニューを表示し、そこでさらに Alt ＋ U （または単に U ）を押す必要があります。しかしショートカットキーを使えば、 Ctrl ＋ U 一発で、いつでも［眉を上げる］メニューを選択できます。

［眉の太さ］というメニューでは、サブメニューを使って眉の太さを選択するようにしてみました。

キャプション	ID
眉毛(&G)	なし
眉を上げる(&U)¥tCtrl+U	ID_BROW_UP
眉を下げる(&D)¥tCtrl+D	ID_BROW_DOWN
眉の太さ	なし

表2-7 [眉毛]メニューのキャプションとメニューID

例によってメニューエディタを起動し、各メニュー項目に**表2-7**のキャプションとメニューIDを与えていきます。なおキャプション中の「¥t」はTABコードを意味します。「¥t」をキャプション中に含むとショートカットキーの表示をメニュー右側にきれいに揃えることができます。

[眉の太さ]メニューではサブメニューを利用するので、プロパティボックスの[ポップアップ]をチェックしてください。するとメニューの横にサブメニュー用のニューアイテムボックスが生じます(**図2-25**)。

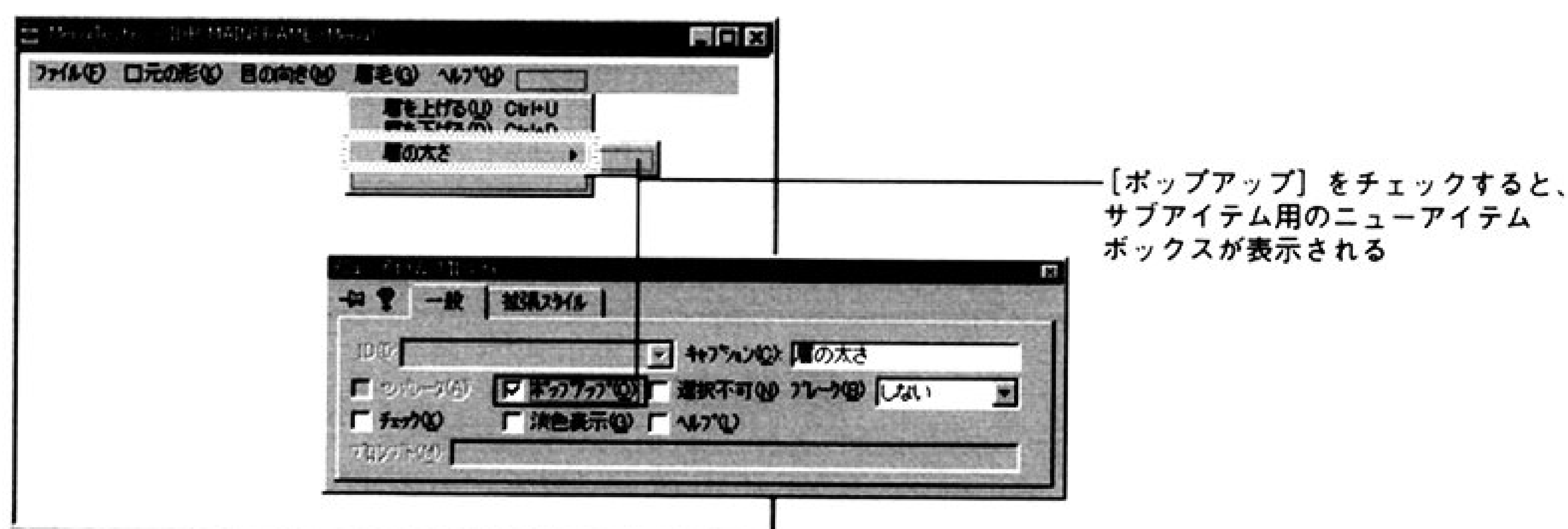


図2-25 [ポップアップ]をチェックしたところ

サブメニューには**表2-8**のキャプションとメニューIDを与えます。

キャプション	ID
1ドット	ID_BROW_1
3ドット	ID_BROW_3
8ドット	ID_BROW_8

表2-8 サブメニューのキャプションとメニューID

これでメニューの外見は完成しました。次にショートカットキーの定義を行います。

ショートカットキーの定義を行うには、まずワークスペースウィンドウの[ResourceView]ページで[Accelerator]をダブルクリックしてください(ショートカットキーは別名「アクセラレータ」ともいう)。すると、リソースブラウザの右の欄に「IDR_MAINFRAME」と

MenuTest.rc - IDR_MAINFRAME (Accelerator)

ID	キー	タイプ
ID_EDIT_COPY	Ctrl + C	VIRTKEY
ID_FILE_NEW	Ctrl + N	VIRTKEY
ID_FILE_OPEN	Ctrl + O	VIRTKEY
ID_FILE_SAVE	Ctrl + S	VIRTKEY
ID_EDIT_PASTE	Ctrl + V	VIRTKEY
ID_EDIT_UNDO	Alt + VK_BACK	VIRTKEY
ID_EDIT_CUT	Shift + VK_DELETE	VIRTKEY
ID_NEXT_PANE	VK_F6	VIRTKEY
ID_PREV_PANE	Shift + VK_F6	VIRTKEY
ID_EDIT_COPY	Ctrl + VK_INSERT	VIRTKEY
ID_EDIT_PASTE	Shift + VK_INSERT	VIRTKEY
ID_EDIT_CUT	Ctrl + X	VIRTKEY
ID_EDIT_UNDO	Ctrl + Z	VIRTKEY

図 2-26 ショートカットキーの一覧

いう ID が表示されるので、これをダブルクリックします。アクセラレータテーブルエディタが起動し、現在設定されているショートカットキーの一覧が表示されます (**図 2-26**)。

このとき表示されるショートカットキーは、AppWizard がスケルトンを作成した時点で定義したもので、MenuTest プロジェクトでは使用しませんから、Delete を何度か押すか、あるいは削除したい項目を右クリックしてショートカットメニューから［切り取り］を選択するなどして、すべて削除してしまってもかまいません (プロジェクトによっては初期設定をそのまま利用することもあり、いつでも削除できるわけではない)。

次に、アクセラレータテーブルエディタの任意の位置を右クリックしてショートカットメニューから［アクセラレータの新規作成］を選んで、ショートカットキーの設定を行うプロパティボックスを開きます。まず［眉を上げる］メニューに Ctrl + U というショートカットキーを割り当ててみましょう。

［ID］ボックスには、ショートカットキーに対応するメニュー項目の ID を指定します。ここでは［眉を上げる］メニューの ID、すなわち［ID_BROW_UP］を入力すればよいわけで

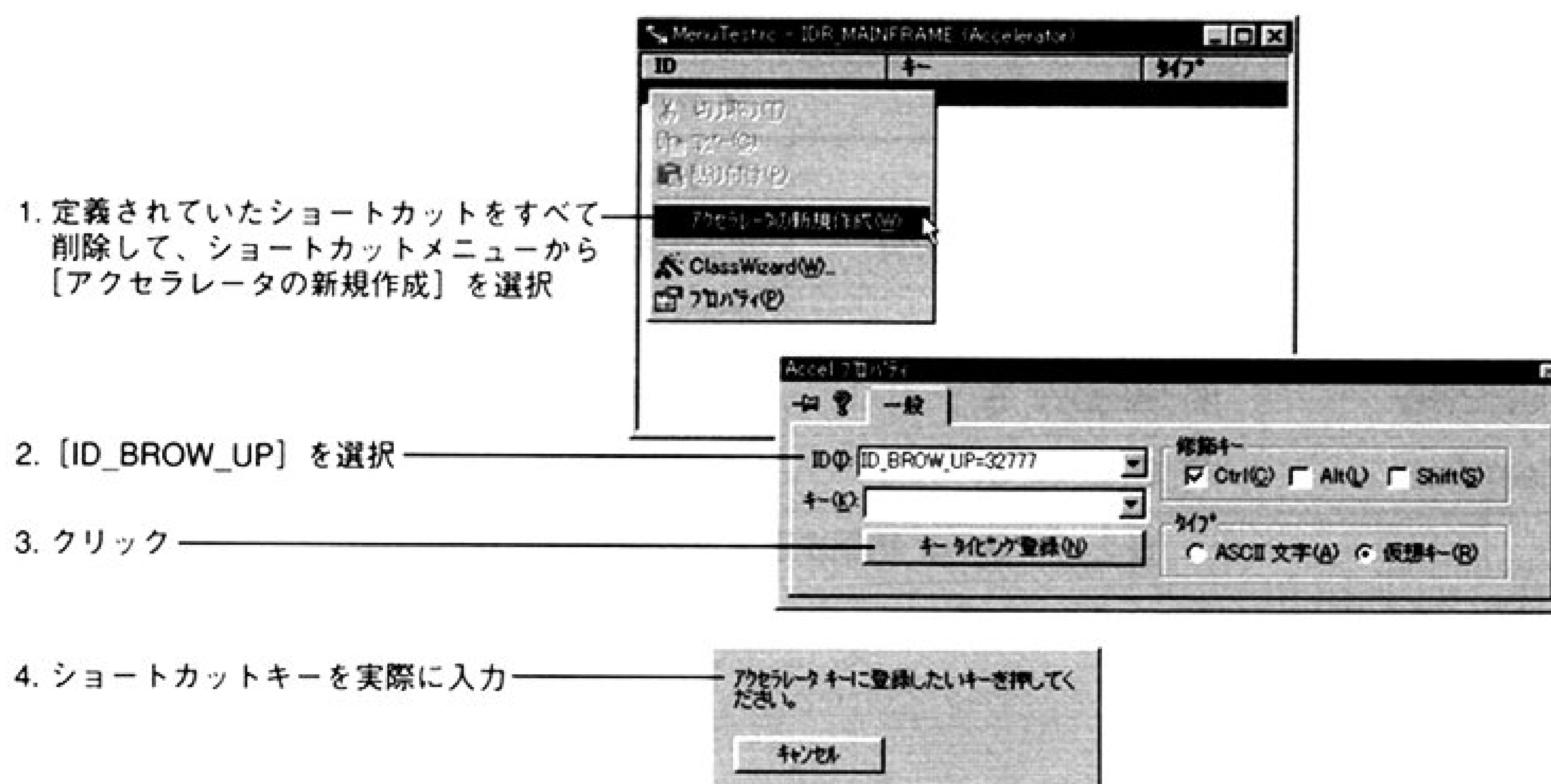


図 2-27 ショートカットキーの設定

す。次に<キータイピング登録>ボタンをクリックすると、ショートカットキーの入力を促すダイアログボックスが開きますから、そこで Ctrl + U を押してください(**図 2-27**)。

Enter キーを押すと登録は完了し、次のショートカットキー定義に移ることができます。いまと同様の手順で、[眉を下げる] メニュー (ID_BROW_DOWN) に、Ctrl + D を割り当ててください。

ショートカットキーには、シフトキー (Shift 、 Ctrl 、 Alt) + その他のキーのほとんどの組み合わせが利用可能です。Shift + Ctrl + X のように、複数のシフトキーを同時に押すこともできます。文字キーだけではなく、ファンクションキーやカーソルキーも使えます。

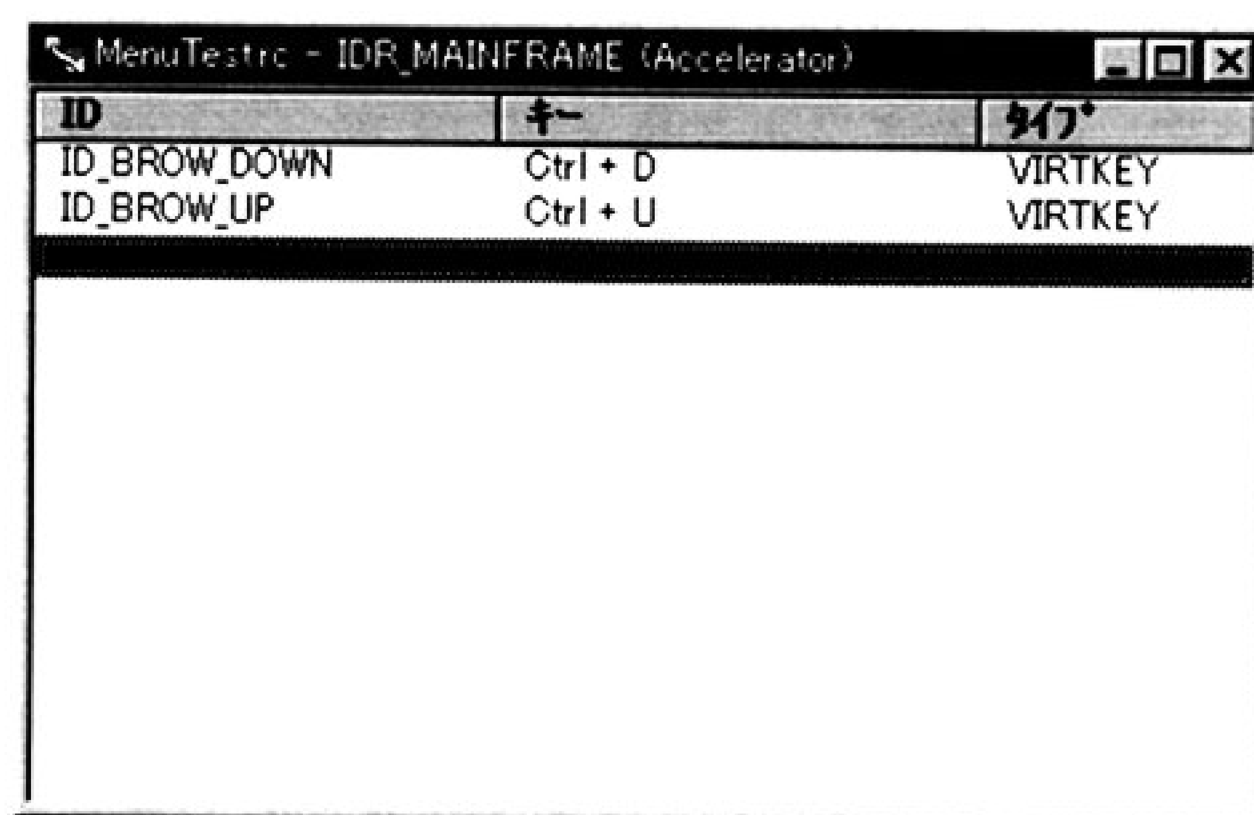


図 2-28　新しく 2 つのショートカットキーを定義したところ

ショートカットキーの設定が終了したら、ClassWizard を起動します。COMMAND メッセージを発生する 5 つのメニューには、それぞれ**リスト 2-9** に示すメッセージハンドラを作成します。メッセージハンドラと ClassWizard ダイアログボックスの各項目との対応は**表 2-9** に示します。

クラス名	オブジェクト名	メッセージ	メッセージハンドラ
CMenuTestView	ID_BROW_UP	COMMAND	OnBrowUp
CMenuTestView	ID_BROW_DOWN	COMMAND	OnBrowDown
CMenuTestView	ID_BROW_1	COMMAND	OnBrow1
CMenuTestView	ID_BROW_3	COMMAND	OnBrow3
CMenuTestView	ID_BROW_8	COMMAND	OnBrow8

表 2-9　眉毛に関係するメッセージハンドラ

リスト2-9 COMMANDメッセージのメッセージハンドラ(MenuTestView.cpp)

```
void CMenuTestView::OnBrowUp()
{
  BrowPos += 10;
  InvalidateRect(NULL, FALSE);
}

void CMenuTestView::OnBrowDown()
{
  BrowPos -= 10;
  InvalidateRect(NULL, FALSE);
}

void CMenuTestView::OnBrow1()
{
  BrowWidth = 1;
  InvalidateRect(NULL, FALSE);
}

void CMenuTestView::OnBrow3()
{
  BrowWidth = 3;
  InvalidateRect(NULL, FALSE);
}

void CMenuTestView::OnBrow8()
{
  BrowWidth = 8;
  InvalidateRect(NULL, FALSE);
}
```

[眉の太さ]のサブメニューの[1ドット]、[3ドット]、[8ドット]という3つのメニュー項目には、**リスト2-10**のようなUPDATE_COMMAND_UIメッセージハンドラを設定して、現在選択されている項目をチェックするようにします。メッセージハンドラの指定は**表2-10**に示します。

クラス名	オブジェクト名	メッセージ	メッセージハンドラ
CMenuTestView	ID_BROW_1	UPDATE_COMMAND_UI	OnUpdateBrow1
CMenuTestView	ID_BROW_3	UPDATE_COMMAND_UI	OnUpdateBrow3
CMenuTestView	ID_BROW_8	UPDATE_COMMAND_UI	OnUpdateBrow8

表2-10 眉毛の太さにチェックマークを付けるメッセージハンドラ

リスト2-10 UPDATE_COMMAND_UI メッセージハンドラ(MenuTestView.cpp)

```
void CMenuTestView::OnUpdateBrow1(CCmdUI* pCmdUI)
{
  pCmdUI->SetCheck((BrowWidth == 1) ? 1 : 0); // 眉の太さが1ならチェック
}

void CMenuTestView::OnUpdateBrow3(CCmdUI* pCmdUI)
{
  pCmdUI->SetCheck((BrowWidth == 3) ? 1 : 0); // 眉の太さが3ならチェック
}

void CMenuTestView::OnUpdateBrow8(CCmdUI* pCmdUI)
{
  pCmdUI->SetCheck((BrowWidth == 8) ? 1 : 0); // 眉の太さが8ならチェック
}
```

また[眉を上げる]と[眉を下げる]の各メニュー項目については、UPDATE_COMMAND_UI メッセージハンドラの中で、CCmdUI::Enable 関数を実行し、有効化／無効化の制御をします(**リスト 2-11**)。この関数は引数が TRUE のときメニューを有効化し、FALSE のとき無効化します。無効化されたメニュー項目は灰色の文字で表示され、選択することができなくなります。

クラス名	オブジェクト名	メッセージ	メッセージハンドラ
CMenuTestView	ID_BROW_UP	UPDATE_COMMAND_UI	OnUpdateBrowUp
CMenuTestView	ID_BROW_DOWN	UPDATE_COMMAND_UI	OnUpdateBrowDown

表2-11 メニューを無効化するメッセージハンドラ

リスト2-11 眉の角度を変更するための UPDATE_COMMAND_UI メッセージハンドラ(MenuTestView.cpp)

```
void CMenuTestView::OnUpdateBrowUp(CCmdUI* pCmdUI)
{
  pCmdUI->Enable((BrowPos < 20) ? TRUE : FALSE); // 眉の位置が20以上なら無効化
}

void CMenuTestView::OnUpdateBrowDown(CCmdUI* pCmdUI)
{
  pCmdUI->Enable((BrowPos > -20) ? TRUE : FALSE); // 眉の位置が-20以下なら無効化
}
```

メッセージハンドラを作成したら、プロジェクトをコンパイルして、ショートカットキーの動作を確認してください。[眉毛]のプルダウンメニューを表示していなくとも、Ctrl+UやCtrl+Dを押すと眉毛の角度が変わるはずです(**図 2-29**)。

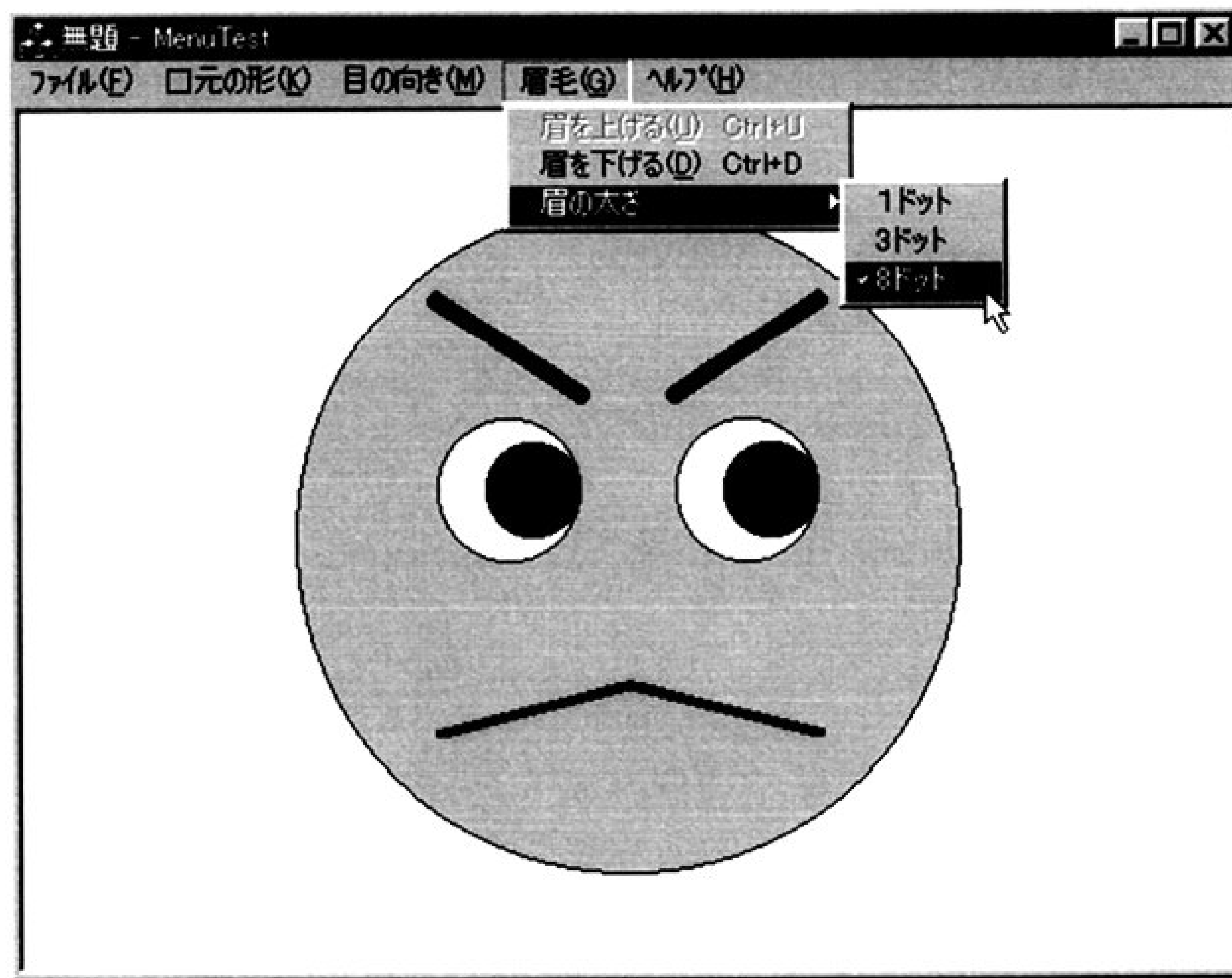


図 2-29 完成した MenuTest プロジェクト

3 ダイアログボックスを使ってみよう

この章ではDlgTestというプロジェクトを作成しながら、ダイアログボックスの使い方について説明をします。このプロジェクトは前章のMenuTestでメニューを使って行った操作をダイアログボックスでの操作に置き換えたものと考えてください。DlgTestプロジェクトのソースは付属CD-ROMの「DlgTest」フォルダに収めてあります。

3.1 スケルトンの作成

まずDlgTestプロジェクトの下準備として、AppWizardでスケルトンを作成し、前章と同様な顔表示ルーチンを追加するところまで作業を進めます。以下はそのプロジェクトの体裁をまとめたものです。

- プロジェクト名：DlgTest
- アプリケーションのタイプ：SDI
- データベースのサポート：しない
- 複合ドキュメントのサポート：しない
- ツールバー／ステータスバー：なし
- 印刷と印刷プレビュー：なし
- そのほか：デフォルトのまま

AppWizardのオプション設定は、MenuTestと同じくSDI形式を使い、［ドッキングツールバー］と［印刷および印刷プレビュー］についてもチェックをはずした状態でスケルトンを作ります。次にリソースエディタを使用して、メニューバーを作成します。AppWizardが生成したメニューは、［ファイル］－［アプリケーションの終了］と、［ヘルプ］－［バージョン情報(DlgTest)］だけ残してあとは削除し、**表3-1**に示すメニュー項目をメニューバーに追加します。

キャプション	ID
口元の形 (&K)	ID_MENU_MOUTH
目の向き (&M)	ID_MENU_EYE
眉毛 (&G)	ID_MENU_BROW

表3-1 DlgTestプロジェクトのメニュー設定

せっかくのダイアログボックスの利用例ですから、ここではプルダウンメニューをまったく使わないことにしましょう。つまり、メニューバーに表示されている項目をクリックすると、COMMANDメッセージが生じ、その結果ダイアログボックスが表示されるようにします。メニューバーに表示されるメニュー項目に関しては、通常はプロパティボックスの［ポップアップ］がチェックされていますが、このチェックを取り除くとプルダウンメニューが表示されなくなり、同時にプロパティボックスの［ID］ボックスが有効になってメニューIDを入力できるようになります。メニューIDを持てるということは、ここをクリックするとCOMMANDメッセージが生じるということです（**図3-1**）。

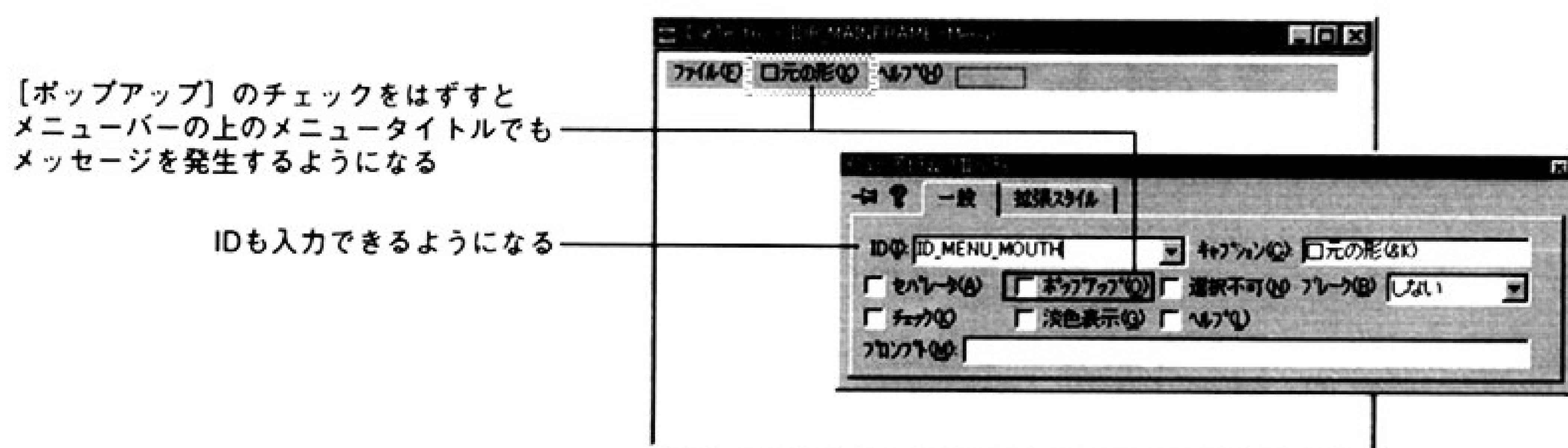


図3-1 ［ポップアップ］のチェックをはずしたところ

ついでに［アプリケーションの終了］と［バージョン情報（DlgTest）］も、それぞれのプルダウンメニューからメニューバーにドラッグ&ドロップを利用して移して、空になった［ファイル］と［ヘルプ］は削除してしまいましょう。最終的にメニューバーは**図3-2**のようになります。



図3-2 このプロジェクトのメニューバー

次にCDlgTestView::OnDraw関数に、「顔」を描くプログラムを記述します。これは前節のMenuTestのCMenuTestView::OnDraw関数とまったく同じ内容ですから、そちらを参照してください。MenuTestView.cppからコピー&ペーストしてしまってもかまいません。

以上で準備はおしまい。次からは、ダイアログボックスの作成手順と、その利用方法を説明します。

3.2 ダイアログボックスの作成

ダイアログボックスを新規に作成するには、メニューから[挿入]－[リソース]を選択して[リソースの挿入]ダイアログボックスで[Dialog]をダブルクリックするか、あるいはプロジェクトワークスペースウィンドウに[ResourceView]ページを表示し、次に[Dialog]フォルダを右クリックしてショートカットメニューから[Dialogの挿入]を選択します。すると**図3-3**のようなダイアログエディタとコントロールパレットが表示されます。

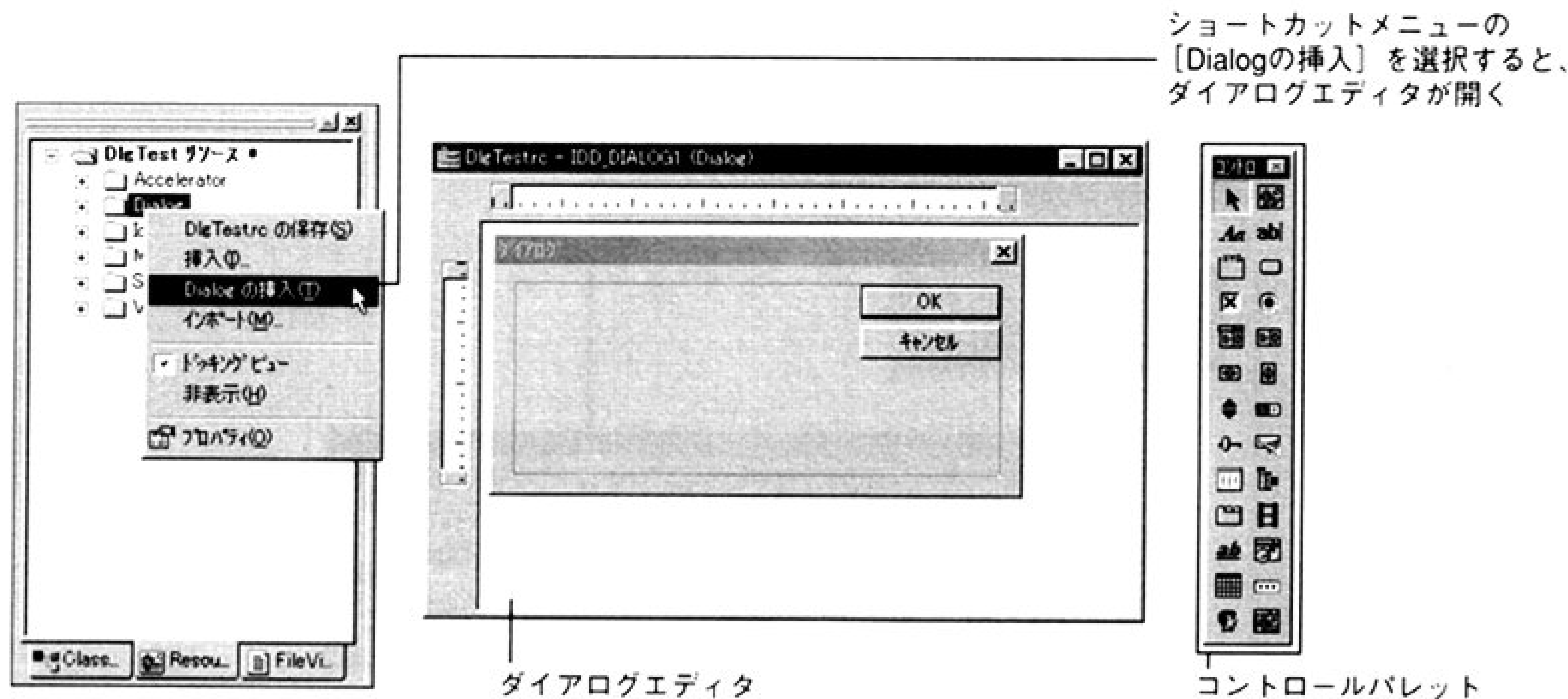


図3-3　ダイアログエディタとコントロールパレット

ダイアログボックスは、土台になるフレームと、フレームの上に配置したいくつかの**コントロール**から構成されます。コントロールとはユーザーインターフェイスの部品となるもので、終了を確認するためのダイアログボックスでは＜OK＞ボタンや＜キャンセル＞ボタンがそうですし、そのほかにもさまざまな種類のコントロールがあります。

ダイアログボックスのサイズや、コントロールの位置とサイズは、変更したい部分をマウスでドラッグすることによって自由に変えられます。またコントロールやダイアログボックスをダブルクリックすると、プロパティボックスが表示され、キャプションやIDが設定できるようになります。新しいコントロールをコントロールパレットで選択し、ダイアログボックス上でマウスをドラッグするだけで追加することもできます。ダイアログエディタを利用すると、このようにコントロールを配置していくだけで、ちょうど貼り絵細工でも作るような感覚で簡単にダイアログボックスを作成することができます。ダイアログエディタの詳しい使い方は、ここでは説明しきれませんので、Visual C++のオンラインマ

ニュアルを参照してください。

DlgTest プロジェクトでは、4種類のダイアログボックスを作ります。まず最初は、プログラムの終了確認に使うダイアログボックスで、<OK>ボタンと<キャンセル>ボタンを備えただけの非常にシンプルなものです。ダイアログエディタが最初に表示するデフォルトのダイアログボックスと同じ構成ですから、そのボタンの位置やサイズを変更するだけであっというまに完成です(**図3-4**)。



図3-4 [終了してもよいですか?]ダイアログボックスの完成図

<OK>ボタンと<キャンセル>ボタンは、プロパティの変更は必要ありません。そしてダイアログボックス自身のプロパティには、**図3-5**のようにデータを設定します。

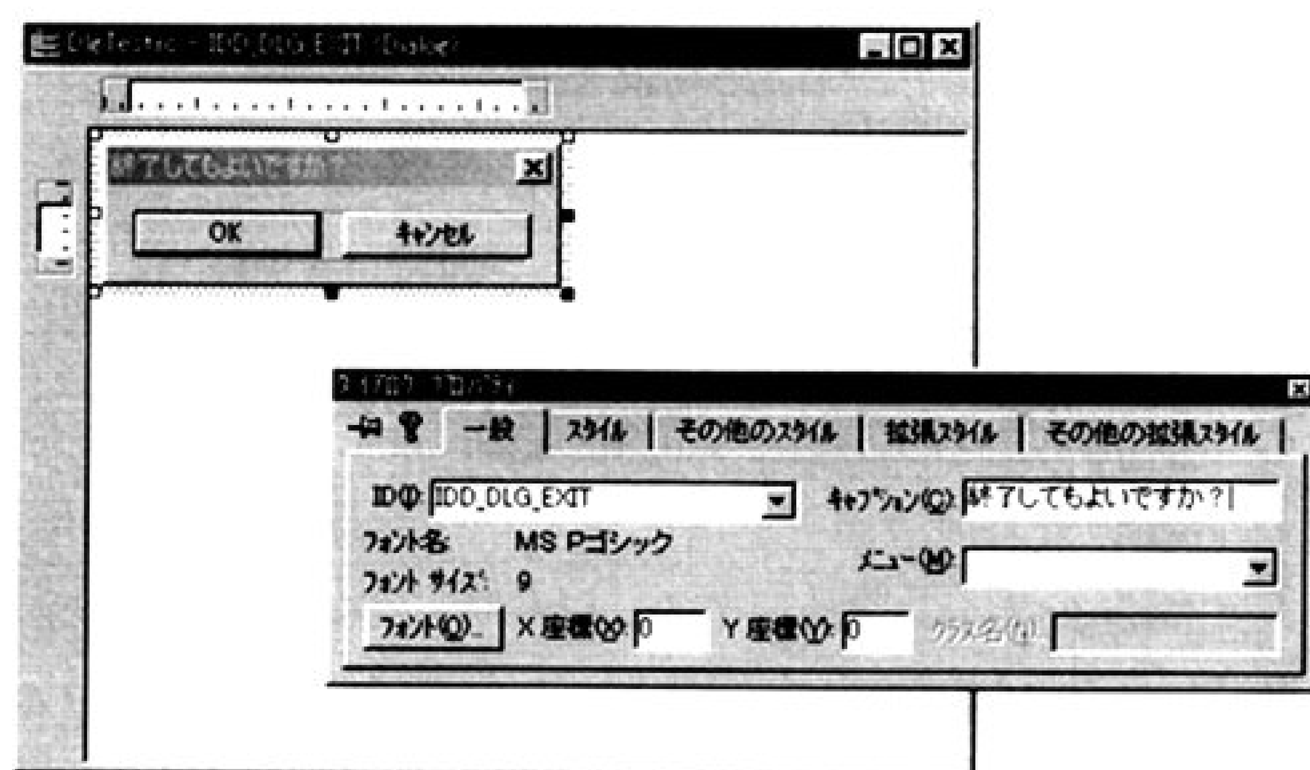


図3-5 [終了確認]ダイアログボックス(IDD_DLG_EXIT)のプロパティ

ダイアログボックスの[キャプション]とは、タイトルバーに表示される文字列のことです。[ID]にはダイアログボックスのリソースIDを指定します。ここは、メニューIDと同様に、名前を指定すればリソースエディタが自動的に適当な値を設定してくれます。このプロジェクトでは、ダイアログボックスのID名はすべて「IDD_DLG」で始めることにしました。

[フォント名]と[フォントサイズ]には、ダイアログボックス内で使用するフォントの名前とサイズが表示されています。初期設定はMS Pゴシックフォントですが、<フォント>ボタンを押すとフォント変更のダイアログボックスが開きます。キャプション(常にSystemフォントを使用)以外のすべての文字は、ここで指定したフォントを用いて表示されます。またダイアログボックスの中では、[フォントの幅の1/4]をx軸方向の1単位、[フォントの高さの1/8]をy軸方向の1単位とする長さの単位(ダイアログ単位)が使わ

れます。ダイアログボックスのサイズや、コントロールの位置とサイズは、すべてダイアログ単位で指定されるため、フォントのサイズを変えるとダイアログボックス全体のサイズも自動的にそれに合わせて調整されます（**図 3-6**）。

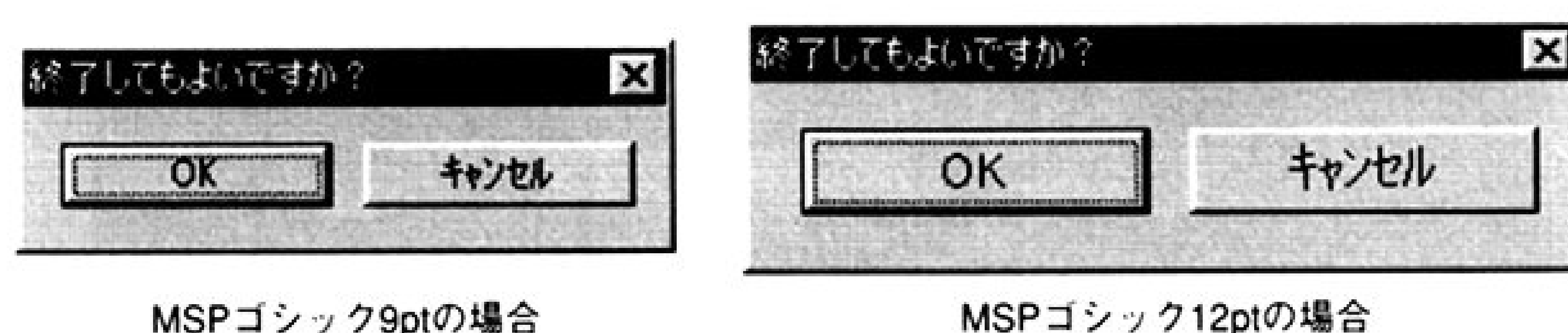


図 3-6　フォントサイズを変更したダイアログボックス

x 座標と y 座標は、ダイアログボックスの表示位置をダイアログ単位で表したものです。ダイアログボックスの親ウィンドウの左上隅が、原点（0, 0）にあたります。

3.3 ダイアログボックスクラスの登録

前章で見たように、メニューエディタで作成したメニューは、すぐにもプログラムの中で利用できました。しかしダイアログボックスの場合は、ダイアログエディタで構造を決めたあとにもう一仕事残っています。このダイアログボックスを扱うための専用のクラスの定義です。この違いは、メニューが単なるウィンドウ上の部品（コントロール）なのに対し、ダイアログボックスは独立したウィンドウとして実現されるからです。一般に Visual C＋＋では、構造の異なるウィンドウごとに、新しいクラスが必要になります。

ダイアログエディタで今作成したダイアログボックスの任意の位置を右クリックして、ショートカットメニューから［ClassWizard］を選択すると ClassWizard が起動します。ダイアログボックスを選択しておいて、メニューから［表示］－［ClassWizard］を選択したり、ツールバーの ClassWizard ボタンをクリックしたり、あるいは Ctrl ＋ W を押してもかまいません。ClassWizard が起動すると、［クラスの追加］ウィンドウが開きます。ここで＜新規クラスの作成＞を選択して＜OK＞ボタンをクリックすると、ダイアログエディタからダイアログボックス ID などのデータが送り込まれ、**図 3-7** に示す設定が表示されます。

［クラス名］には新しいクラスの名前を入力します。このとき名前の先頭を大文字の“C”で始めると、MFC の他のクラス名との一貫性が保て、プログラムが読みやすくなります。ここでは CEndDlg クラスと名付けました。

ダイアログボックスのクラスは、すべて CDialog クラスから派生したものでなければなりません。そのため［基本クラス］には［CDialog］を指定するわけですが、ダイアログエディタから ClassWizard を起動した場合は、この欄には自動的に CDialog が選択されます。

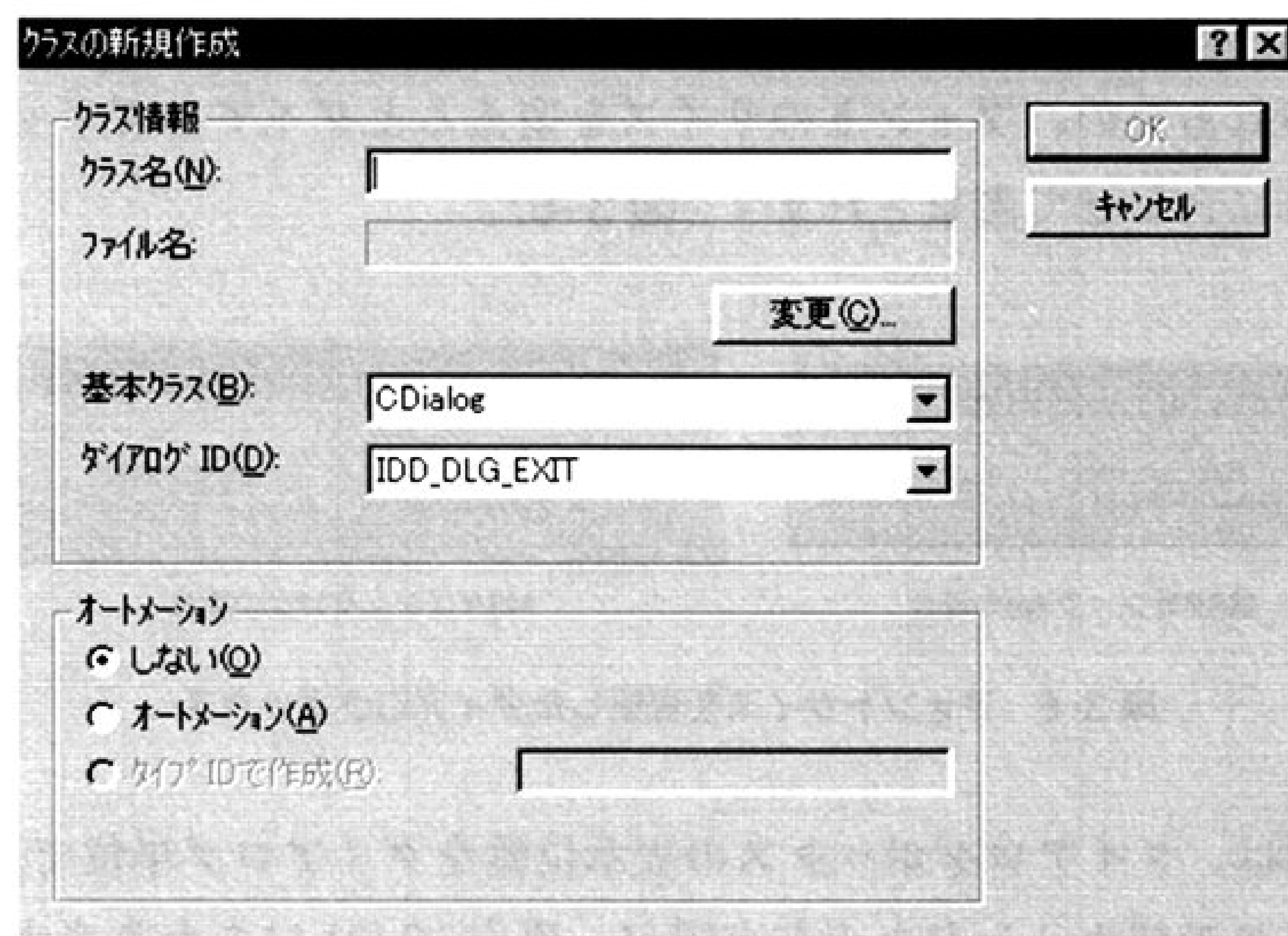


図 3-7 ［クラスの新規作成］ダイアログボックス

［ダイアログ ID］には、ダイアログエディタで指定したダイアログボックスのリソース ID を入力する必要がありますが、これもダイアログエディタで指定したリソース ID で自動的に設定されます。

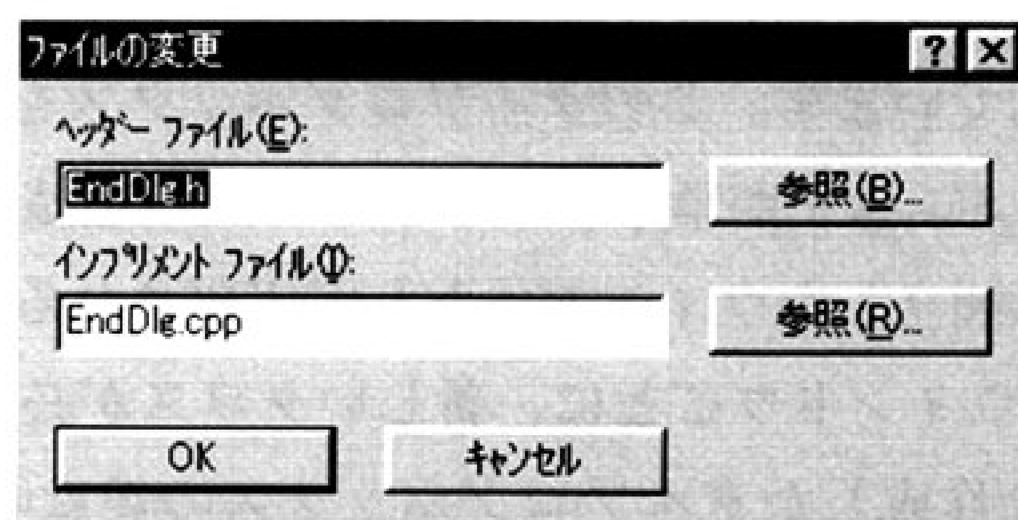


図 3-8 ファイル名の指定

［ファイル］には、［クラス名］に指定したダイアログボックスクラスの名前をもとに適当な実装ファイルのファイル名が表示されます。ヘッダファイルはここに表示されている実装ファイルの「.cpp」の部分が「.h」に変わったものです。<変更>ボタンをクリックすると、ヘッダファイルと実装ファイルのファイル名をプログラマが指定することも可能です（**図 3-8**）。プログラマが実装ファイルおよびヘッダファイルのファイル名を指定するときに、すでにプロジェクトに存在するファイル名を指定した場合、クラスの定義とその実装は指定したファイルの最後に付け足されます。ここでは ClassWizard が表示したファイル名をそのまま使用することとします。

［オートメーション］では、OLE オートメーションクライアントにこのクラスを公開したり、このクラスのオブジェクトをクライアントが作成できるように指定しますが、ここでは OLE は関係ないので［しない］をオンにしておけばよいでしょう。

設定がすべて終わったら、<OK>ボタンをクリックします。すると**リスト3-1**と**リスト3-2**に示す2つのファイルが作成され、プロジェクトに追加されます。

リスト3-1 ヘッダーファイル(EndDlg.h)

```
#if !defined(AFX_ENDDLG_H__1CE62927_2EE5_11D2_ACD8_00A0247DB2AD__INCLUDED_)
#define AFX_ENDDLG_H__1CE62927_2EE5_11D2_ACD8_00A0247DB2AD__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000
// EndDlg.h : ヘッダー ファイル
//

/////////////////////////////////////////////////////////////////////////////
// CEndDlg ダイアログ

class CEndDlg : public CDialog
{
// コンストラクション
public:
  CEndDlg(CWnd* pParent = NULL);   // 標準のコンストラクタ

// ダイアログ データ
  //{{AFX_DATA(CEndDlg)
  enum { IDD = IDD_DLG_EXIT };
    // メモ: ClassWizard はこの位置にデータ メンバを追加します。
  //}}AFX_DATA


// オーバーライド
  // ClassWizard は仮想関数のオーバーライドを生成します。
  //{{AFX_VIRTUAL(CEndDlg)
  protected:
  virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV サポート
  //}}AFX_VIRTUAL

// インプリメンテーション
protected:

  // 生成されたメッセージ マップ関数
  //{{AFX_MSG(CEndDlg)
    // メモ: ClassWizard はこの位置にメンバ関数を追加します。
  //}}AFX_MSG
  DECLARE_MESSAGE_MAP()
};

//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ は前行の直前に追加の宣言を挿入します。

#endif//!defined(AFX_ENDDLG_H__1CE62927_2EE5_11D2_ACD8_00A0247DB2AD__INCLUDED_)
```

リスト3-2 インプリメントファイル(EndDlg.cpp)

```
// EndDlg.cpp : インプリメンテーション ファイル
//

#include "stdafx.h"
#include "DlgTest.h"
#include "EndDlg.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////
// CEndDlg ダイアログ

CEndDlg::CEndDlg(CWnd* pParent /*=NULL*/)
  : CDialog(CEndDlg::IDD, pParent)
{
  //{{AFX_DATA_INIT(CEndDlg)
    // メモ-ClassWizardはこの位置にマッピング用のマクロを追加または削除します。
  //}}AFX_DATA_INIT
}


void CEndDlg::DoDataExchange(CDataExchange* pDX)
{
  CDialog::DoDataExchange(pDX);
  //{{AFX_DATA_MAP(CEndDlg)
    // メモ-ClassWizardはこの位置にマッピング用のマクロを追加または削除します。
  //}}AFX_DATA_MAP
}


BEGIN_MESSAGE_MAP(CEndDlg, CDialog)
  //{{AFX_MSG_MAP(CEndDlg)
    // メモ-ClassWizardはこの位置にマッピング用のマクロを追加または削除します。
  //}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////
// CEndDlg メッセージ ハンドラ
```

CEndDlgクラスはCDialogクラスの機能をほとんどそのまま受け継いだだけのクラスです。したがって、EndDlg.cppには最初は特筆するようなコードはまったく含まれていません。CEndDlgクラス独自の動作が必要なときは、このEndDlg.cppでメンバ関数のオーバーライドを行います。

EndDlg.h には CEndDlg クラスの定義が記述されています。CEndDlg クラス独自のメンバ変数が必要な場合は、このクラス定義を修正します。またプロジェクトの他のファイル中で CEndDlg クラスを利用する際には、この EndDlg.h をインクルードしなければなりません。

3.4 ダイアログボックスの表示

CDialog クラスから派生した CEndDlg などのクラスには、DoModal 関数というメンバ関数があり、これを利用すれば簡単にダイアログボックスを表示することができます。具体的な手順は次のとおりです。

```
CEndDlg dlg;   // CEndDlg クラスのオブジェクトを定義する
int retcode;

retcode = dlg.DoModal();   // dlg の DoModal 関数を実行する
```

DoModal 関数を実行すると、同一プログラム内の他のウィンドウはアクセス不可能になり、表示されたダイアログボックスにしかアクセスできなくなります。適当な操作を行ってからダイアログボックスを閉じると、DoModal 関数は終了し、返り値を返します。

このようにアプリケーションの他の部分を一時停止させて動作するダイアログボックスを、**モーダル（Modal）形式**と呼びます。Windows では、このほかに**モードレス（Modeless）形式**といって、アプリケーションと並列に動作するダイアログボックスも利用することができますが、モードレスダイアログボックスを実現するプログラムは少々複雑なので、本書では扱いません。

DoModal 関数の返り値はダイアログボックスの終了ステータスを示します。たとえば、<OK>ボタンを押して終了した場合は IDOK、<キャンセル>ボタンならば IDCANCEL という返り値になります。なお、DoModal 関数に任意の返り値を返させる方法は次節で説明します。

それでは、この CEndDlg クラスのダイアログボックスを利用して、プログラムを終了するとき、本当に終了してよいか確認を求める機能を DlgTest プロジェクトに組み込んでみましょう。

［アプリケーションの終了］メニューを選択すると、ID_APP_EXIT というオブジェクト ID を持つ COMMAND メッセージが発生しますから、それをメッセージハンドラで捕まえます。メッセージはアプリケーションクラス（CDlgTestApp クラス）で処理すればよいでしょう。つまり ClassWizard で**表 3-2** に示す指定をしてメッセージハンドラを作成すればよいわけです。

クラス名	CDlgTestApp
オブジェクト ID	ID_APP_EXIT
メッセージ	COMMAND
関数名	OnAppExit

表 3-2 ［終了］メニューのメッセージハンドラの指定

ClassWizard が出力したスケルトンは、**リスト 3-3** のように変更します。この関数は、表示したダイアログボックスが<OK>ボタンで閉じられた場合に限り、CWinApp::OnAppExit 関数を実行してプログラムを終了させます。CEndDlg クラスを利用するためには、DlgTest.cpp に EndDlg.h をかならずインクルードしてください。

リスト 3-3 変更した CDlgTestApp::OnAppExit 関数（DlgTest.cpp）

```
#include "EndDlg.h"                  // かならずインクルードすること

void CDlgTestApp::OnAppExit()
{
  CEndDlg dlg;

  if (dlg.DoModal() == IDOK) {  // ダイアログボックスを表示し
    CWinApp::OnAppExit();       // <OK>ボタンが押されたらプログラムを終える
  }                             // <OK>ボタンが押されなければ何もしない
}
```

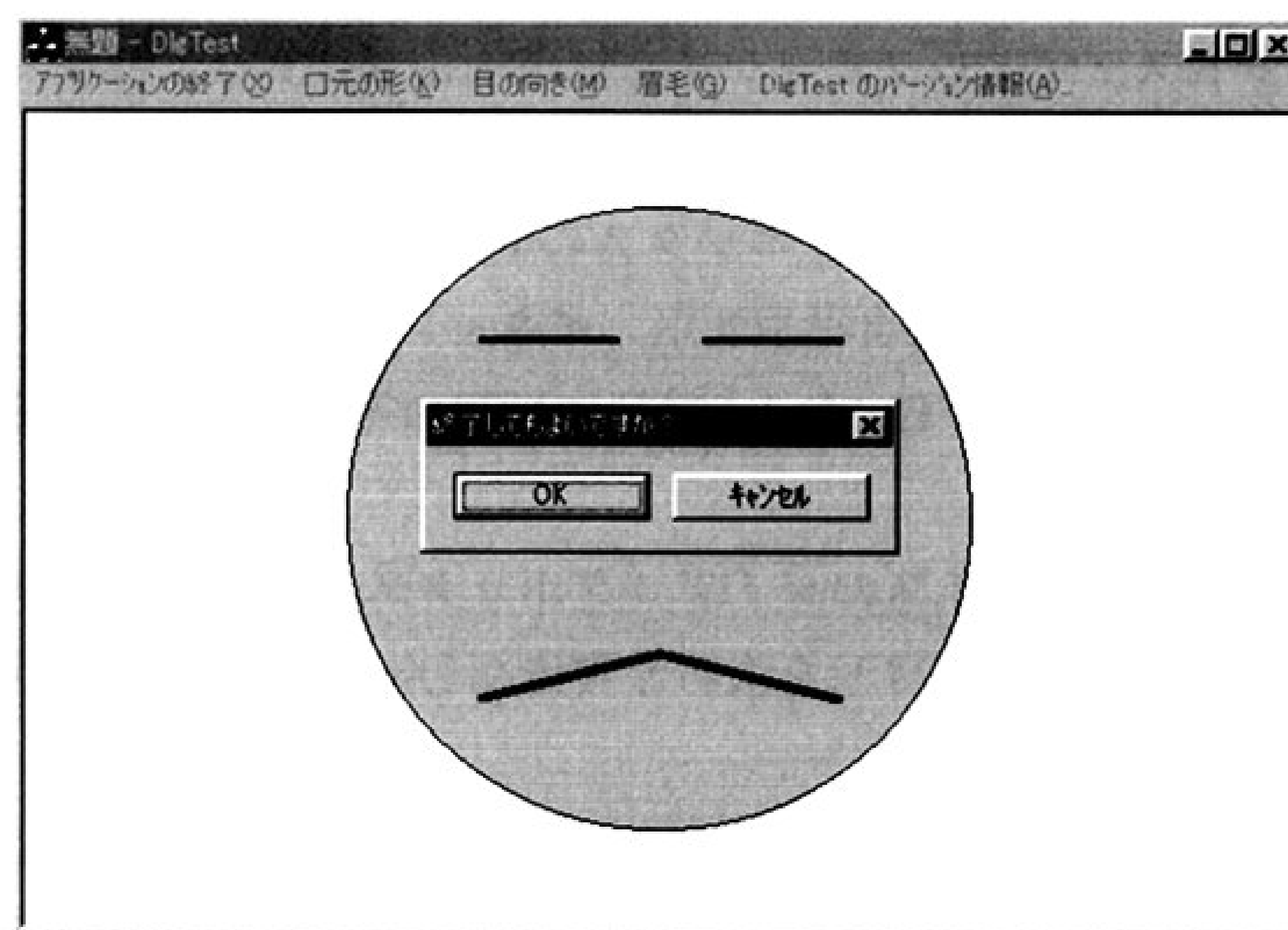


図 3-9 実行画面

なお、この場合 EndDlg.cpp にはまったく手を加える必要はありません。

3.5 終了ステータスを知るには？

ダイアログボックスは、原則としてアプリケーションとは別個に動作する独立したウィンドウです。ダイアログボックスと、その上に配置されたボタンなどのコントロールは、一緒になって閉じた世界を作っているといってもよいでしょう。

たとえば、前章のMenuTestの場合、メニュー項目を選択すると選択されたメニュー項目を反映するように画面を再描画していましたが、このときメニューが発生したCOMMANDメッセージはビュークラスのオブジェクトが処理していました。これは、ビュークラスのオブジェクトはメニューで発生したメッセージを受け取り、処理することができたからでした。一方、ダイアログボックス上のボタンは**BN_CLICKED**メッセージを発生しますが、このメッセージを受け取ることができるのは、そのボタンを所有しているダイアログボックスだけなのです*1。ダイアログボックスで画面に変化を与えるようなボタンが押されても、画面を再描画するはずのビューオブジェクトはそのボタンが発生したメッセージを受け取ることができません。ということは、ダイアログボックスでのユーザーの選択結果を取得するための何らかの方法が必要だということです。

ダイアログボックスからデータを取得するのにもっとも簡単な方法は、前節でもいったように、DoModal関数の返り値、すなわちダイアログボックスの終了ステータスを利用することです。DoModal関数で開始したダイアログボックスは、終了するときにかならずCDialog::EndDialog関数を実行します。<OK>ボタンをクリックしてダイアログボックスを閉じた場合は、そのメッセージハンドラCDialog::OnOK関数の中で、EndDialog(IDOK)を実行しています。その結果としてDoModal関数はIDOKを返すのです。

これを利用すると、<OK>や<キャンセル>以外のボタンでも押されたかどうかが簡単に調べられます。ここでは**図3-10**のようなダイアログボックスで、<にっこり>ボタンをクリックすると画面の顔がほほ笑み、<むっつり>ボタンをクリックするとムッとし、<とりやめ>ボタンの場合は何も変わらないというプログラムを作ってみましょう。

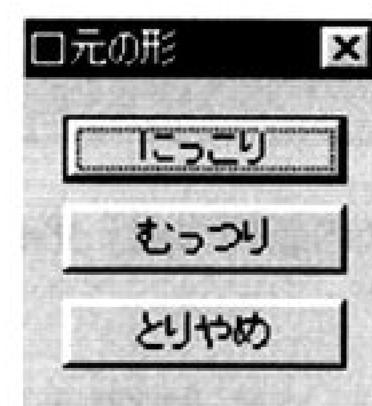


図3-10　[口元の形] ダイアログボックス

*1　BN_CLICKEDメッセージは通知メッセージという。実際には「メッセージ返送」機構によって、ダイアログボックス以外のウィンドウでも通知メッセージを受け取ることができるが、特殊な用途に限られるので、ここでは解説しない。

まず、ダイアログエディタを起動し、3つのプッシュボタンを持つダイアログボックスを作成してください。ダイアログボックスと、それぞれのプッシュボタンには、次のようなキャプションとIDを与えます。<OK>ボタンと<キャンセル>ボタンを流用してもかまいませんが、そのときにはキャプションとIDをプロパティボックスで忘れずに変更してください(**表 3-3**)。

	キャプション	ID
ダイアログボックス	口元の形	IDD_DLG_MOUTH
プッシュボタン1	にっこり	IDC_MOUTH_SMILE
プッシュボタン2	むっつり	IDC_MOUTH_UNHAPPY
プッシュボタン3	とりやめ	IDC_MOUTH_NOCHANGE

表 3-3 [口元の形] ダイアログボックスのキャプションと ID

次に、メニューから[表示]-[ClassWizard]を選択するか、あるいはダイアログボックスを右クリックしてショートカットメニューから[ClassWizard]を選択して、ClassWizardを起動し、ダイアログボックスクラスを追加します。クラス名は[CMouthDlg]とします(**図 3-11**)。

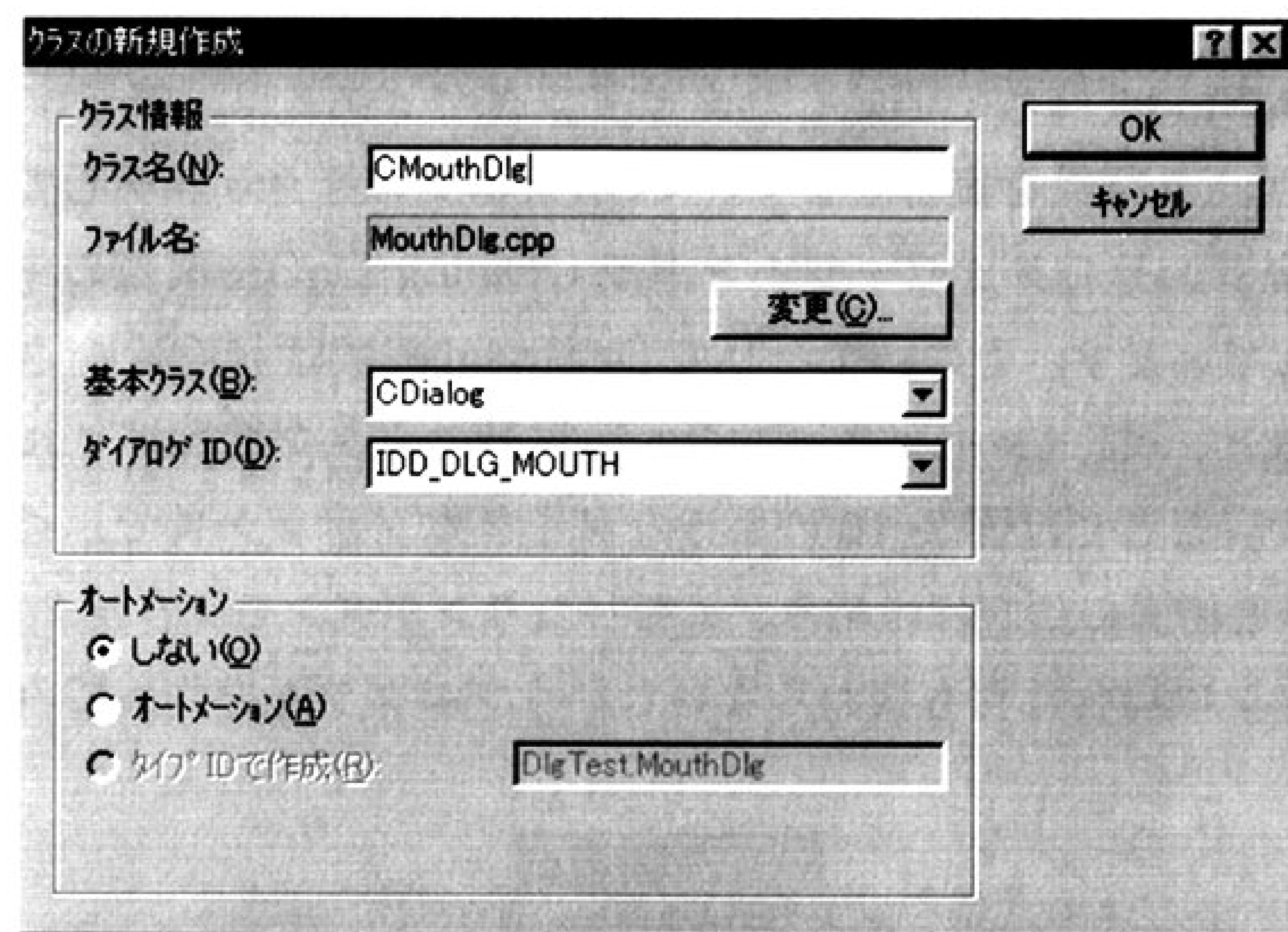


図 3-11 [クラスの新規作成] ダイアログボックス

次にそれぞれのボタンからのメッセージを処理するハンドラを作成します。**リスト 3-4**に示すように、どれも単にメッセージを受け取ったらダイアログボックスを終了させるだけですが、<にっこり>、<むっつり>、<とりやめ>の各ボタンのオブジェクトIDが終了ステータスとして渡されるようになっています。メッセージハンドラ作成時の指定は例によって**表 3-4**に示します。

クラス名	オブジェクト名	メッセージ	メッセージハンドラ
CMouthDlg	IDC_MOUTH_SMILE	BN_CLICKED	OnMouthSmile
CMouthDlg	IDC_MOUTH_UNHAPPY	BN_CLICKED	OnMouthUnhappy
CMouthDlg	IDC_MOUTH_NOCHANGE	BN_CLICKED	OnMouthNochange

表3-4 各プッシュボタンをクリックしたときに実行されるメッセージハンドラ

リスト3-4 ダイアログボックスのメッセージハンドラ（MouthDlg.cpp）

```
void CMouthDlg::OnMouthSmile()
{
  EndDialog(IDC_MOUTH_SMILE);      // 返り値は IDC_MOUTH_SMILE
}

void CMouthDlg::OnMouthUnhappy()
{
  EndDialog(IDC_MOUTH_UNHAPPY);    // 返り値は IDC_MOUTH_UNHAPPY
}

void CMouthDlg::OnMouthNochange()
{
  EndDialog(IDC_MOUTH_NOCHANGE);   // 返り値は IDC_MOUTH_NOCHANGE
}
```

最後に［口元の形］ダイアログを表示する関数、つまり［口元の形］メニューをクリックしたときのメッセージハンドラを作成します。このハンドラは、上記のダイアログボックスを表示し、終了ステータスによって処理を振り分けます（**リスト3-5**）。ClassWizardで、**表3-5**のように指定をしてメッセージハンドラを作成してください。

クラス名	CDlgTestView
オブジェクトID	ID_MENU_MOUTH
メッセージ	COMMAND
関数名	OnMenuMouth

表3-5 ダイアログを表示するメッセージハンドラ

リスト3-5 CDlgTestView::OnMenuMouth関数の実装（DlgTestView.cpp）

```
#include "MouthDlg.h"

void CDlgTestView::OnMenuMouth()
{
  CMouthDlg dlg;
```

```
  switch (dlg.DoModal()) {  // ダイアログを表示
  case IDC_MOUTH_SMILE:       // DoModal 関数が IDC_MOUTH_SMILE を返せばにっこり
    MouthPos = 10;
    InvalidateRect(NULL, FALSE);
    break;
  case IDC_MOUTH_UNHAPPY:
    MouthPos = -10;           // DoModal() が IDC_MOUTH_UNHAPPY を返せばむっつり
    InvalidateRect(NULL, FALSE);
    break;
  default:                    // さもなければ何もしない
    break;
  }
}
```

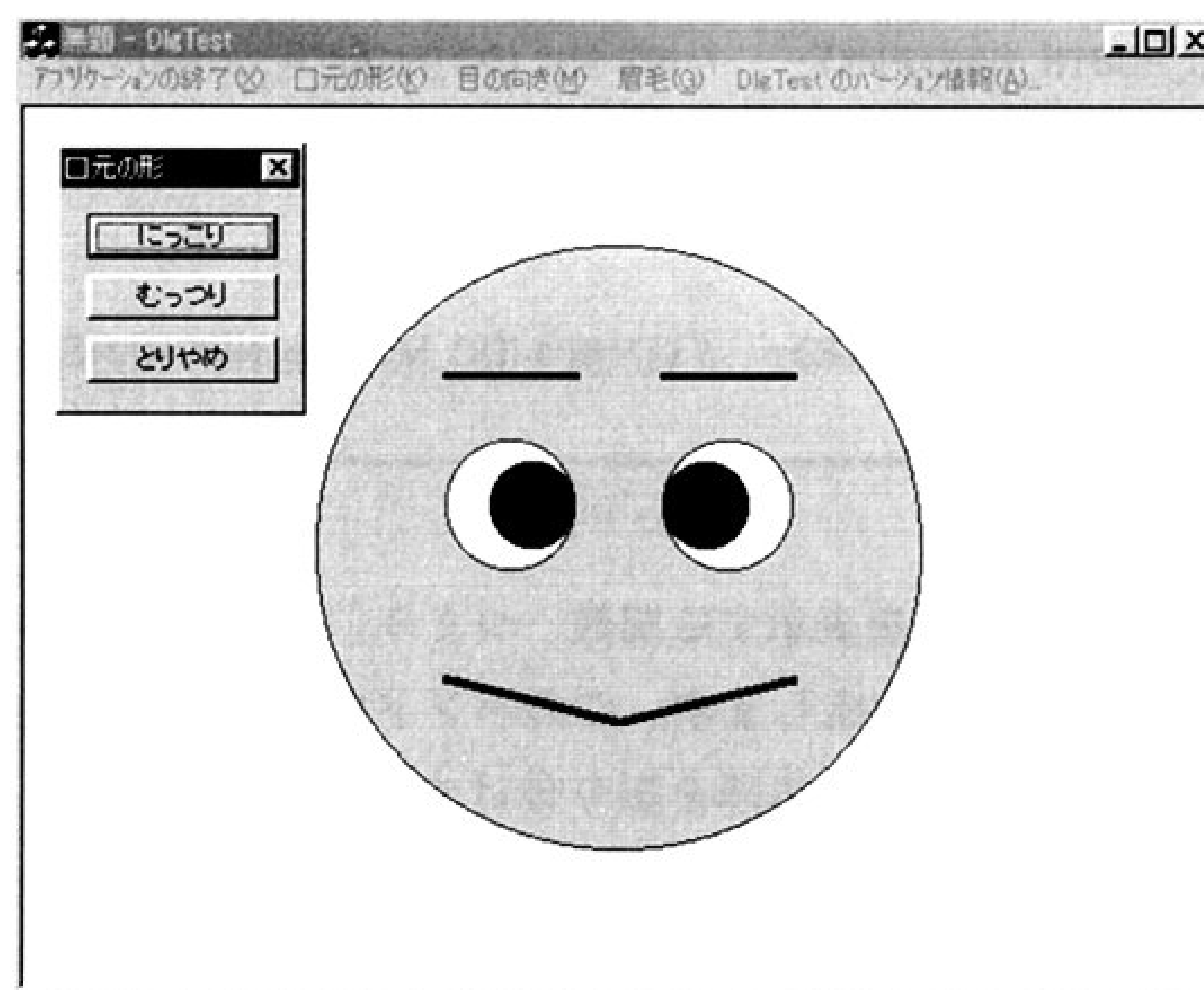


図 3-12 実行画面

3.6 ダイアログボックスの初期化

MenuTest の解説で、メニューにチェックマークを表示したり、メニューを無効化するには、メニューから送られる UPDATE_COMMAND_UI メッセージを処理すればよいことを説明しました。ダイアログボックスの場合も同様に、画面表示の前に何らかの設定を行いたいことが多々あります。もちろんそれを実現する方法も用意されています。ただしメニューの初期化とは少しばかり方法が異なり、こちらは **WM_INITDIALOG** というメッセージを利用します。

WM_INITDIALOGメッセージは、ダイアログボックスが表示される直前にWindowsからダイアログボックスに送られるメッセージで、それを処理するのはダイアログボックス自身です。このようにして、ダイアログボックスは、表示される前に自分自身の初期化を行います。

ここでは、前節で作成したダイアログボックスに、WM_INITDIALOGメッセージを処理するメッセージハンドラを付け加えてみましょう。

クラス名	CMouthDlg
オブジェクトID	なし(CMouthDlgを選択)
メッセージ	WM_INITDIALOG
関数名	OnInitDialog

表3-6 ダイアログの初期化を行うメッセージハンドラの指定

メッセージハンドラの作成には、やはりClassWizardを使用します。このメッセージを受け取るのはCMouthDlgクラスです。メッセージの送り手はWindowsシステムに決まっていますから、オブジェクトIDは必要ありません。メッセージの種類は当然ながらWM_INITDIALOGです。というわけで、ClassWizardで**表3-6**に示す指定にそってメッセージハンドラを作成してください。

ClassWizardが出力するメッセージハンドラのスケルトンは、**リスト3-6**のようになります。

リスト3-6 メッセージハンドラのスケルトン(MouthDlg.cpp)

```
BOOL CMouthDlg::OnInitDialog()
{
  CDialog::OnInitDialog();

  // TODO: この位置に初期化の補足処理を追加してください

  return TRUE;
}
```

ダイアログボックスの標準的な初期化処理は、CDialog::OnInitDialog関数で行われているので、どんなダイアログボックスも省略することはできません。CMouthDlgなどのCDialogクラスの派生クラスのOnInitDialog関数は、かならずその先頭でCDialog::OnInitDialog関数を実行する必要があります。

また、OnInitDialog関数の中でフォーカスを移動した場合、返り値としてFALSEを返さなければなりません。ここでもしTRUEを返すと、ダイアログボックスが実際に表示される時点で、デフォルトのフォーカス位置に戻されてしまいます。

ここではCMouthDlg::OnInitDialog関数の動作として、<とりやめ>ボタンにフォーカスを移動させてみます。コードを**リスト3-7**に、実行画面を**図3-13**に示します。

リスト3-7 CMouthDlg::OnInitDialog関数の実装(MouthDlg.cpp)

```
BOOL CMouthDlg::OnInitDialog()
{
  CDialog::OnInitDialog();

  GetDlgItem(IDC_MOUTH_NOCHANGE)->SetFocus(); // <とりやめ>ボタンにフォーカス移動
  return FALSE;
}
```

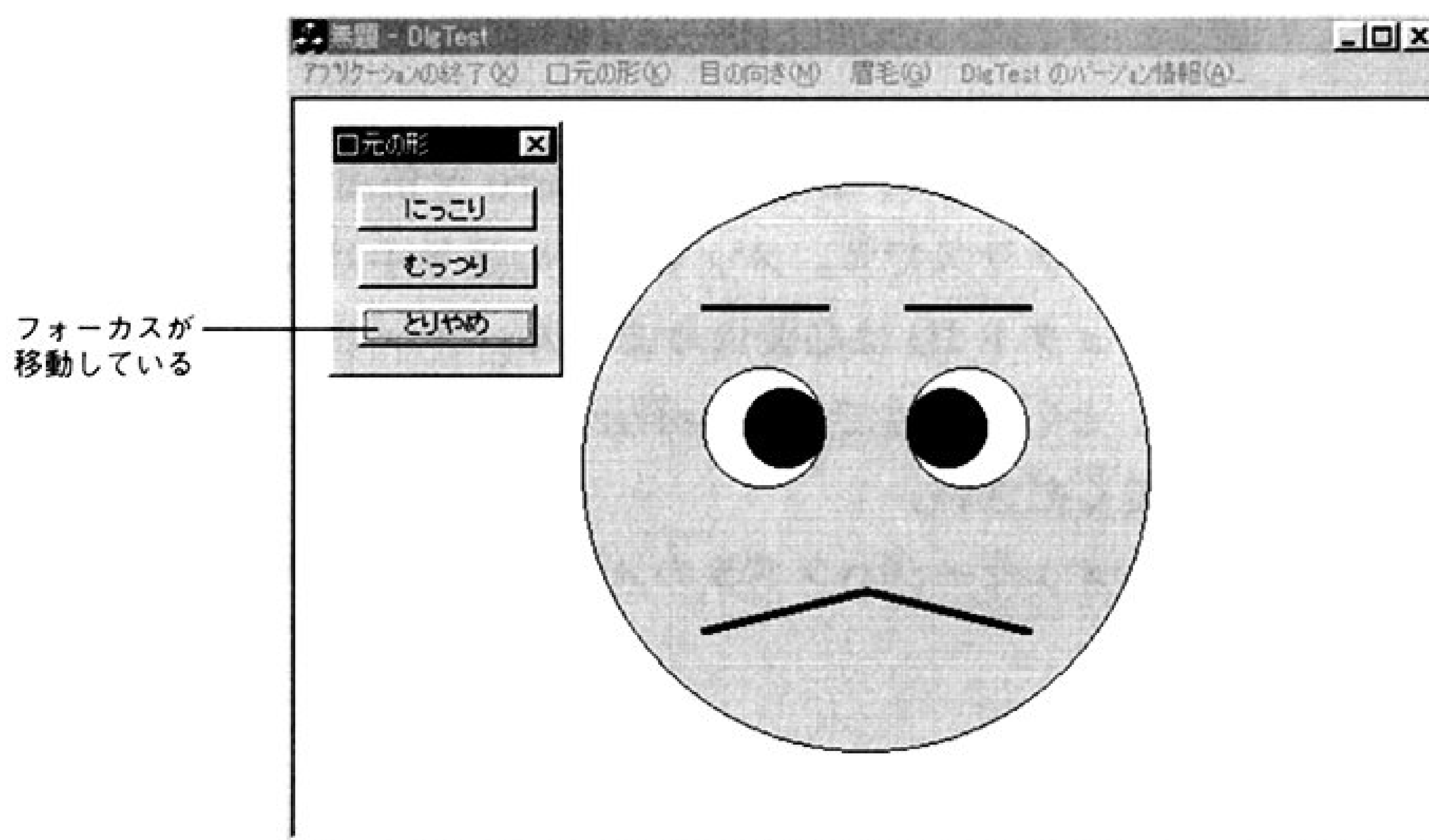


図3-13 実行画面

このメッセージハンドラでは、CWnd::GetDlgItem関数を利用して、引数に指定したID(リソースエディタで設定した値)を持つコントロールへのポインタを取得し、CWnd::SetFocus関数でそのコントロールへフォーカスを移しています。しかし、CWnd::GetDlgItem関数を利用するこの手法はVisual C++ではやや異色のものであり、使用するのはあまりほめられたことではありません(ということにしておいてください)。本来ならばすべての処理は、MFCという美しい(?)クラス階層の中で行われるべきだからです。そこでVisual C++には、ある機能が設けられました。……次節へ続く。

3.7 DDXを利用する

ここまで述べたように、ダイアログボックスへのアクセスは、簡単ではないか、あるいはVisual C++のコンセプトからはずれた"きたない"処理になりがちです。DoModal関数の返り値を利用してダイアログボックスから情報を得る方法も、押されたボタンを判別するくらいなら役に立ちますが、文字列データや複数のコントロールの設定状況など、複雑なデータを得るには不十分です。

そこで、Visual C++では、**DDX(Dialog Data eXchange)**という手段を用いて、この問題に解答を与えました。DDXとは要するに、コントロールの状況を反映するメンバ変数をダイアログボックスを定義するクラスの中に用意し、そのメンバ変数をアクセスすることで、まるでダイアログボックス上のコントロールに直接アクセスしているように見せかけてしまう方法です。

DDXの利用例として、前節で作成したCMouthDlg::OnInitDialog関数を、もっとエレガントなコードに直してみましょう。

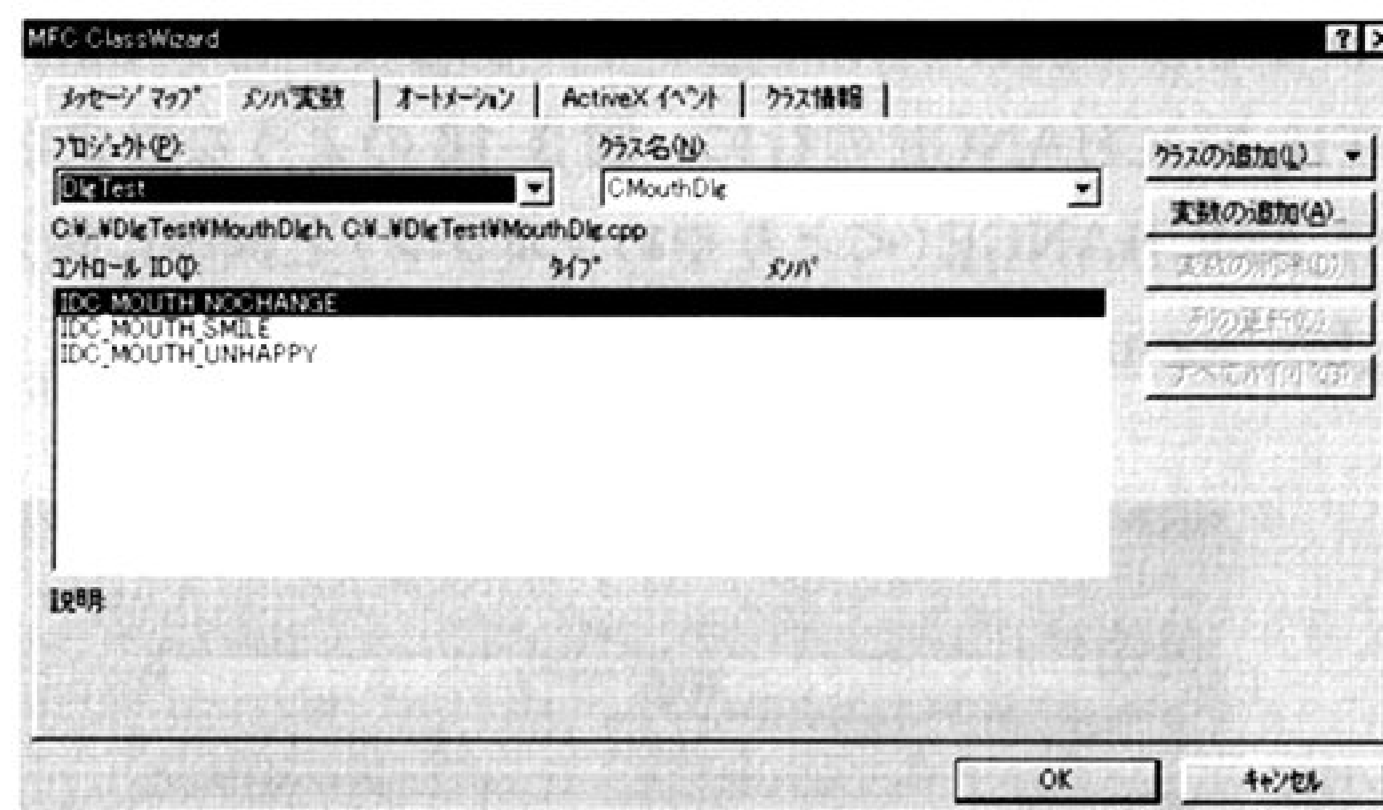


図3-14 [メンバ変数]ページにコントロールが一覧されたところ(メンバ変数追加の前)

まずClassWizardを起動して、[メッセージマップ]ページで[クラス名]から[CMouthDlg]を選んでください。そして、[メンバ変数]タブをクリックしてください。あるいは、[メンバ変数]タブをクリックし[メンバ変数]ページを表示してから、[クラス名]に[CMouthDlg]を選択してもかまいません。そうすると、[メンバ変数]ページにCMouthDlgクラスが定めるダイアログボックスに配置されたコントロールの一覧が表示されます(**図3-14**)。

そこで「IDC_MOUTH_NOCHANGE」を選択して、<変数の追加>ボタンをクリックすると、**図3-15**のような[メンバ変数の追加]ダイアログボックスが開きますから、[メンバ変数]に適当な変数名を入力してください。ここでは「m_btnNochange」とでもしておきましょう。変数名を入力したら<OK>ボタンをクリックします。なお、Visual C++

ではクラスのメンバ変数の名前は、"member" を意味する "m_" から始めることを推奨しています。

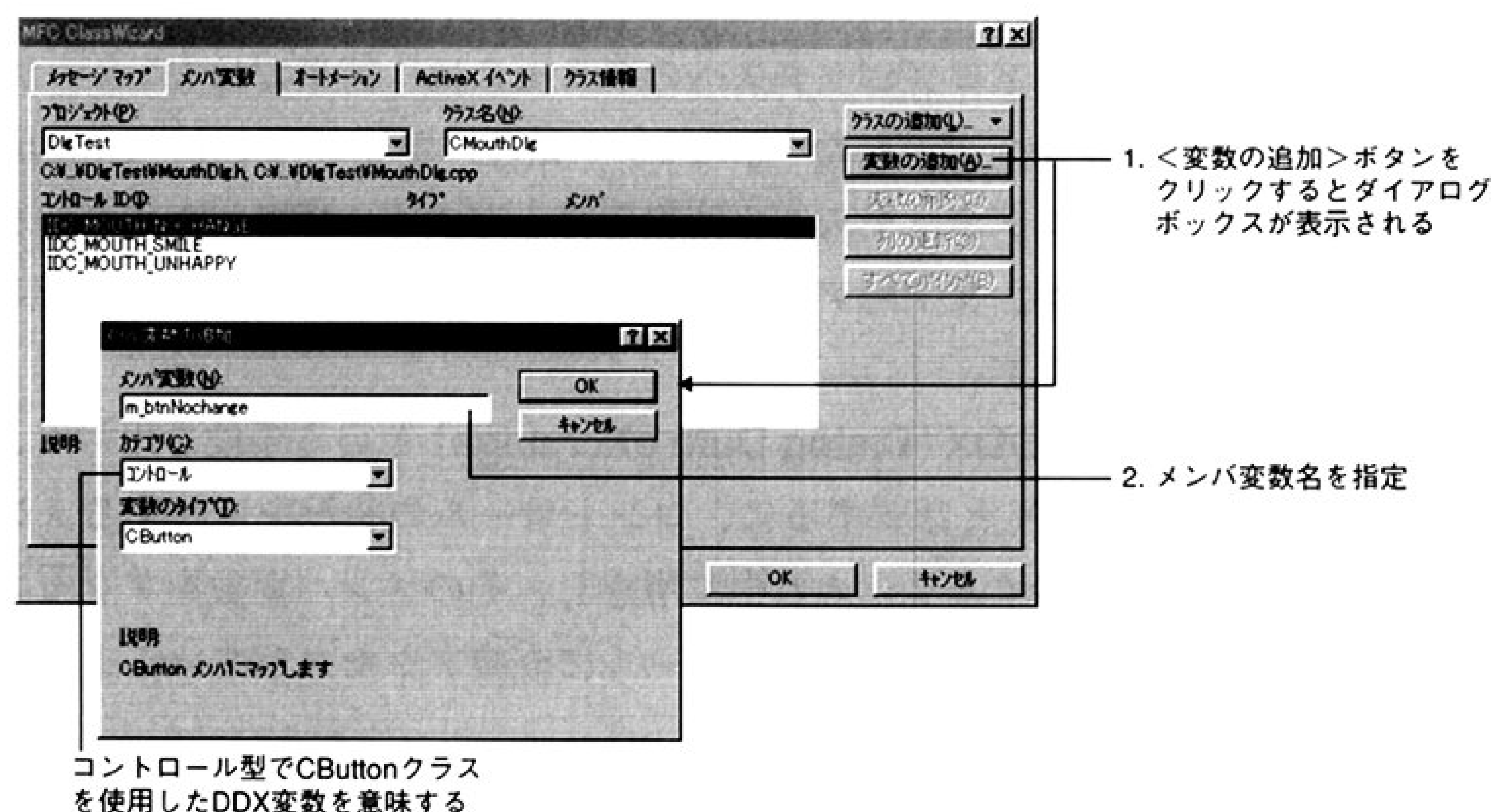


図 3-15 ［メンバ変数の追加］ダイアログボックス

変数名を入力し終わったら<OK>ボタンをクリックして、再び ClassWizard に戻ってみると、IDC_MOUTH_NOCHANGE の行に、**図 3-16** のような設定が書かれています。これは IDC_MOUTH_NOCHANGE（<とりやめ>ボタン）と同等に扱える CButton クラスのメンバ変数、m_btnNochange が作成されたことを意味します。

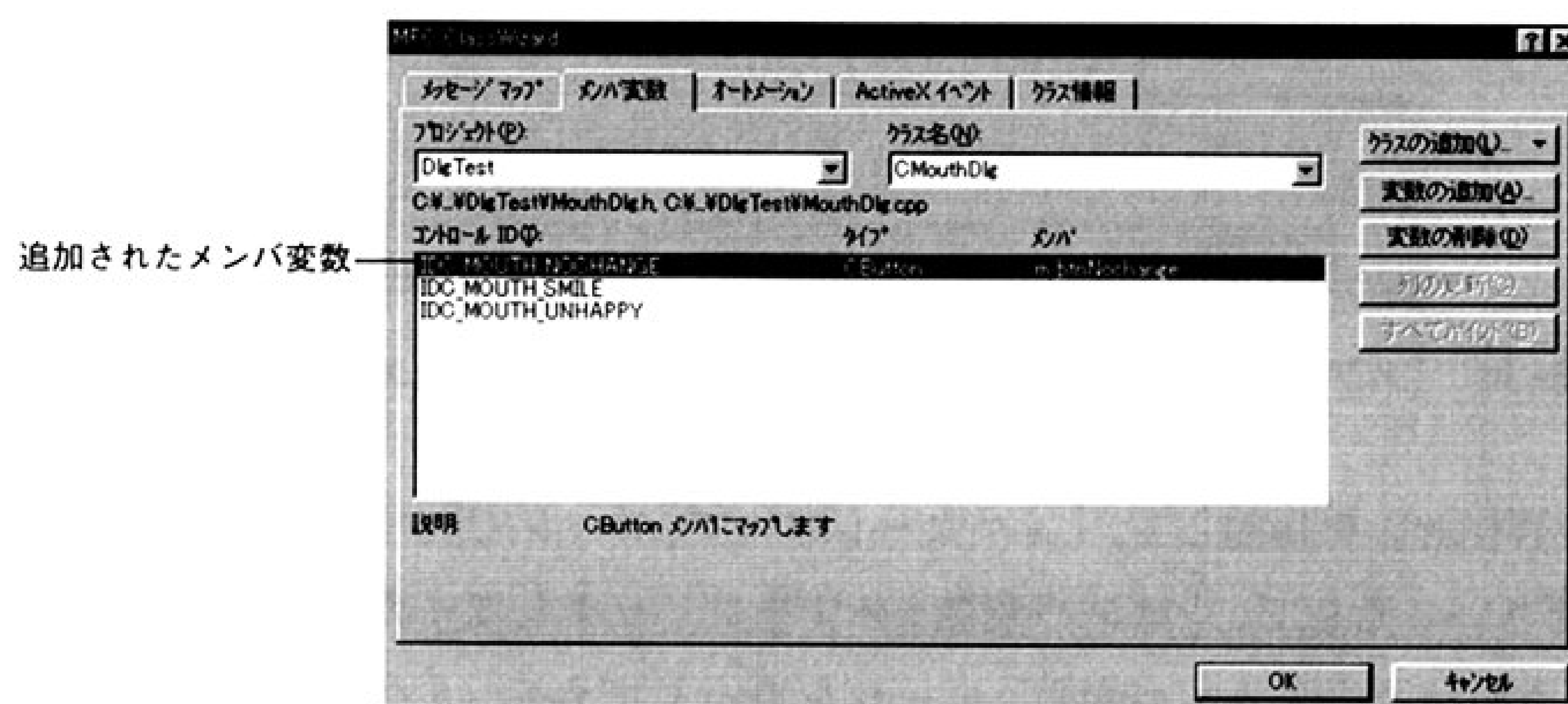


図 3-16 再び ClassWizard（メンバ変数追加のあと）

この m_btnNochange は、CMouthDlg クラスのメンバ変数です。ためしにヘッダファイル MouthDlg.h をのぞいてみましょう。**リスト 3-8** のように、CMouthDlg クラスの定義の中に、m_btnNochange というメンバ変数が作られていることがわかります（灰色で少し見にくいですが…）。

リスト3-8 CMouthDlgのクラス定義(MouthDlg.h)

```
class CMouthDlg : public CDialog
{
// コンストラクション
public:
  CMouthDlg(CWnd* pParent = NULL);   // 標準のコンストラクタ

// ダイアログ データ
  //{{AFX_DATA(CMouthDlg)
  enum { IDD = IDD_DLG_MOUTH };
  CButton m_btnNochange;        //←これだ!!
  //}}AFX_DATA
  ...
}
```

これ以降、m_btnNochangeへのアクセスは、<とりやめ>ボタンのアクセスと同じ意味を持ちます。そこで、m_btnNochangeを使ってCMouthDlg::OnInitDialog関数を書き直せば、**リスト3-9**のようになります。

リスト3-9 変更したCMouthDlg::OnInitDialog関数(MouthDlg.cpp)

```
BOOL CMouthDlg::OnInitDialog()
{
  CDialog::OnInitDialog();

  m_btnNochange.SetFocus(); // <とりやめ>ボタンにフォーカスを移動
  return FALSE;
}
```

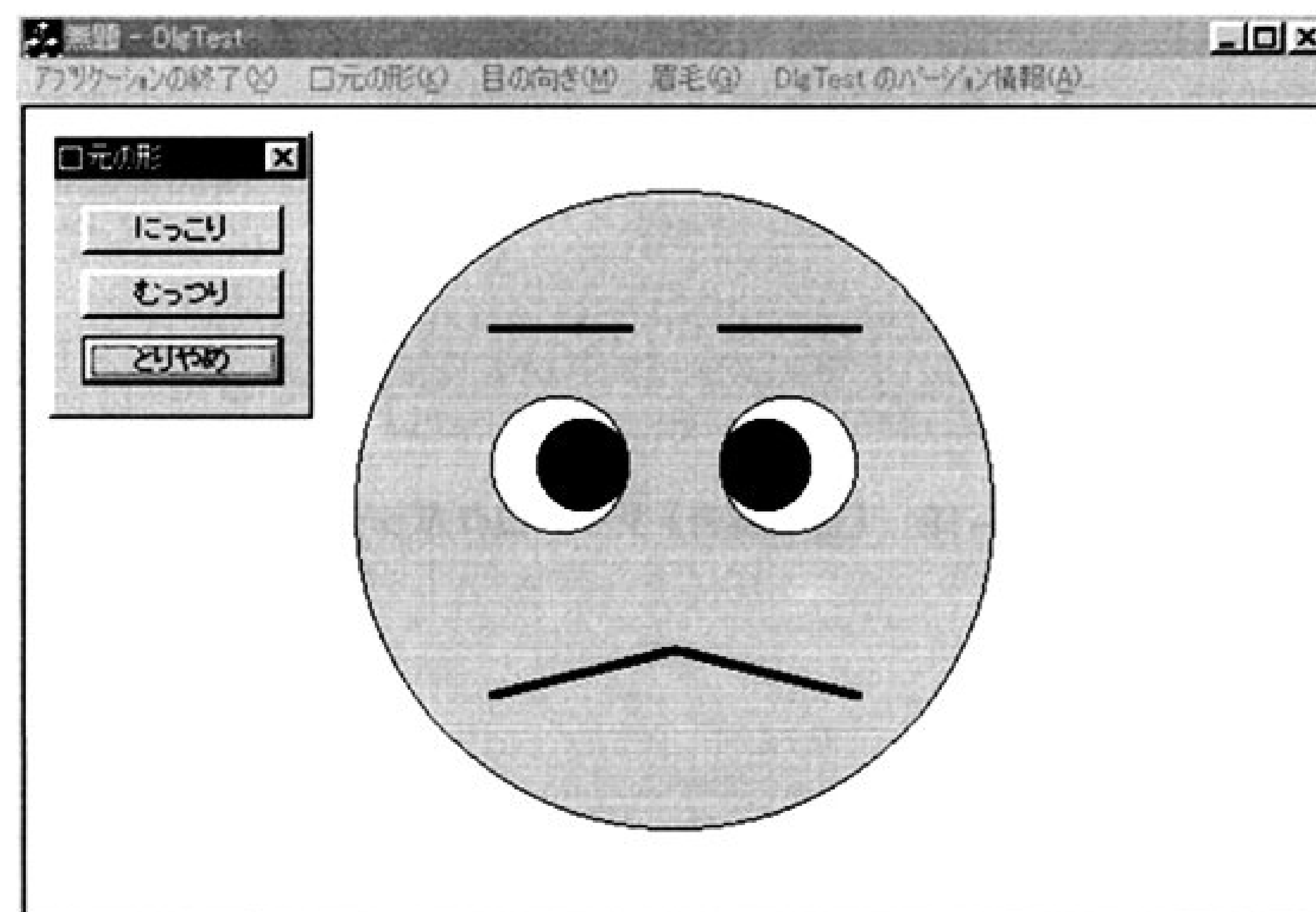


図3-17 実行画面

ここで注意しておきたいのは、m_btnNochangeはボタンの状態を反映する変数にすぎず、ボタンそのものではないということです。あくまでも、DDXの機能が働いて「コントロールを仮想的に表す変数」と「コントロールそのもの」の間で、自動的にデータを交換してくれているため、両者が同等にみなせるのです。

このデータ交換が行われるしくみと、実際にデータが交換されるタイミングについては、次節で説明しましょう。

3.8 値型のDDX

前節で説明したのは、ダイアログボックス上のボタンを、ボタンを表すCButtonクラスの変数を介してアクセスするDDXでした。単純明快ではありますが、CButtonクラスの動作を知らなければ使えないのが難点かもしれません。このように、コントロールをそれと同じクラスの変数で表すDDXを**コントロール型のDDX**と呼びます。

そしてもう1つ、**値型のDDX**と呼ばれるものがあります。これを使うと、コントロールの状態が整数や文字列などの形式で表されるため、より一層扱いやすいものになります。ただし、コントロールの種類によって、利用できるデータ形式は決まっています。

値型のDDXの利用例として、**図3-18**のようなダイアログボックスを準備しましょう。このダイアログボックスは左右の目玉の向きを変えるものです。コントロールの違いによりデータの種類が変わることを見てもらうため、右目の制御にチェックボックス、左目にはラジオボタンという異なるコントロールを使用してみました。

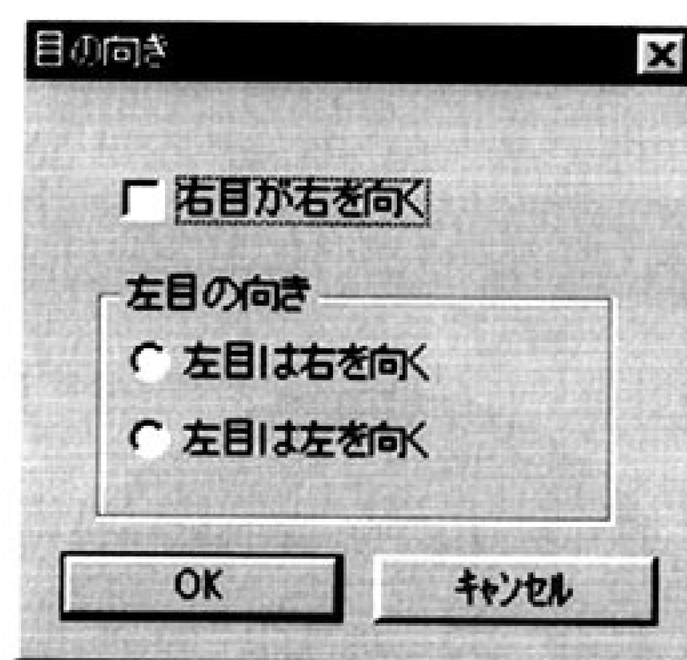


図3-18 ［目の向き］ダイアログボックス

まずダイアログエディタでダイアログボックスを新規作成し、**図 3-19** のような位置に<OK>ボタンと<キャンセル>ボタンを移動します。ダイアログボックスのキャプションは［目の向き］、ID 名は［IDD_DLG_EYE］とします。

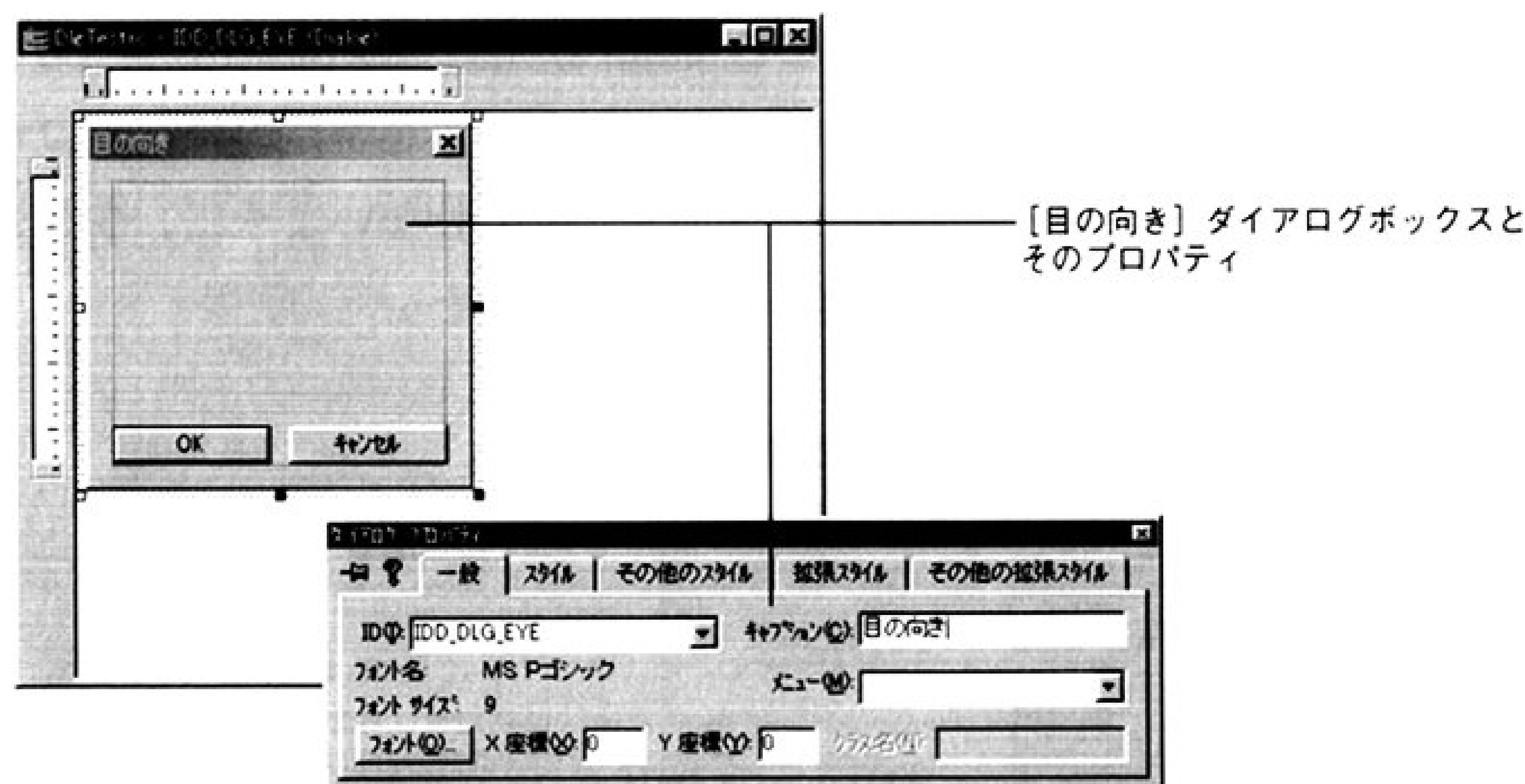


図 3-19　［目の向き］ダイアログボックス（IDD_DLG_EYE）のプロパティ

次にチェックボックスを 1 つ配置し、**図 3-20** のようにプロパティを設定してください。チェックボックスのプロパティのうち、［スタイル］ページの［自動］が最初から選択されていることに注意してください。この設定は変更してはいけません。チェックを取り消してしまうと、［右目が右を向く］チェックボックスをクリックしたときにチェックマークのオン／オフを自力で行わなければならず、プログラミングが非常にやっかいになります。



図 3-20　［右目が右を向く］チェックボックス（IDC_CHK_RR）のプロパティ

その下に2つのラジオボタンを配置します。ラジオボタンもやはり［スタイル］ページの［自動］はかならずチェックされたままにしておいてください。ラジオボタンのIDの値は、1つのグループの中で連続させておくと後々の扱いが楽になります。実際にはラジオボタンを連続して配置すれば、ダイアログエディタが割り当てるIDに割り当てる値（ID値）も連続しますが、確実に連続させるために、ここでは「ID名＝ID値」という形式で、ID名と同時にID値も入力します。このID値は、リソースエディタが他のコントロールに自動的に割り当てる値（100から始まって1ずつ増える）と衝突しないように、少し大きめの値として300および301を選びます（**図3-21**、**図3-22**）。

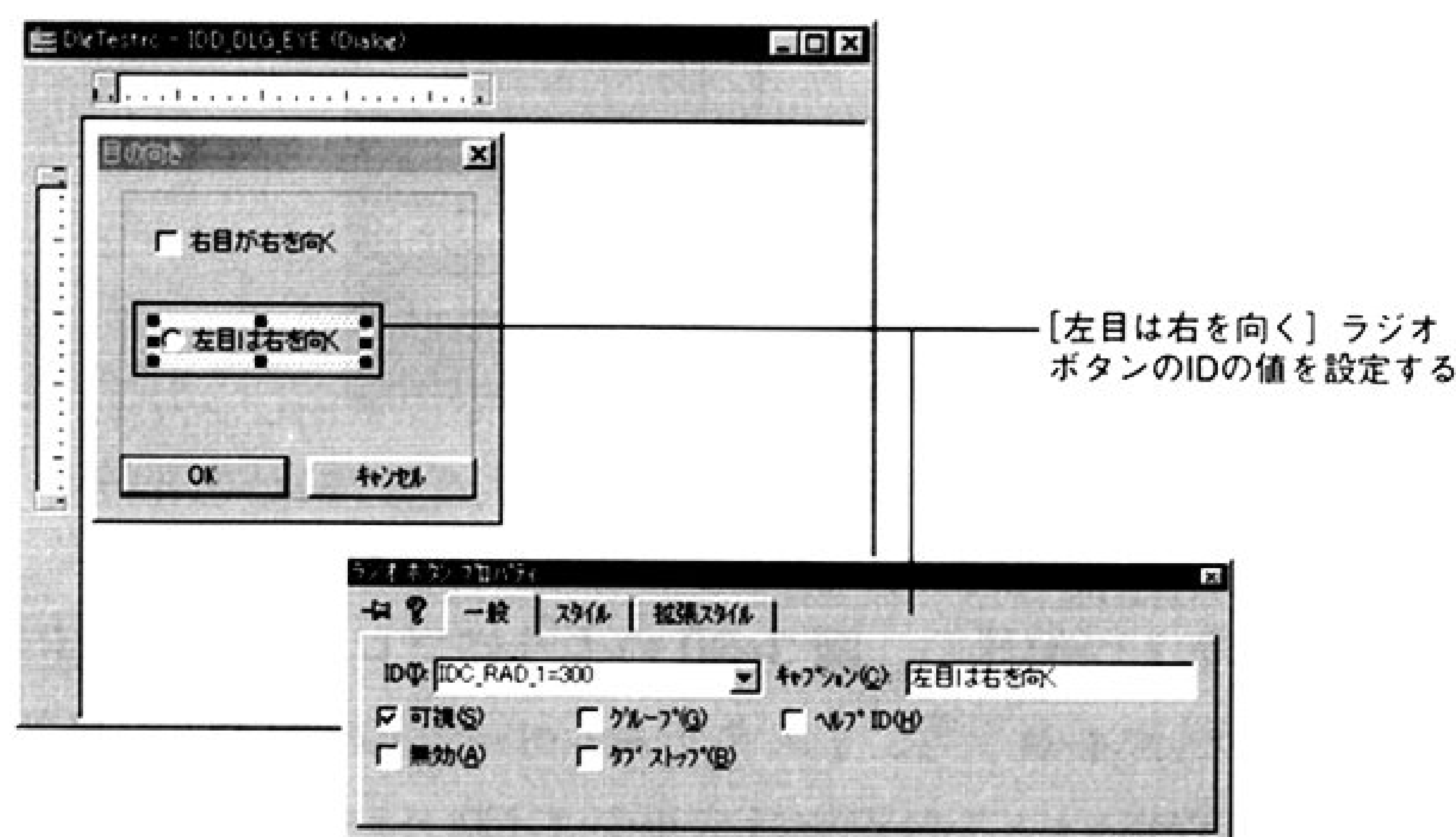


図3-21　［左目は右を向く］ラジオボタン（IDC_RAD_1）のプロパティ

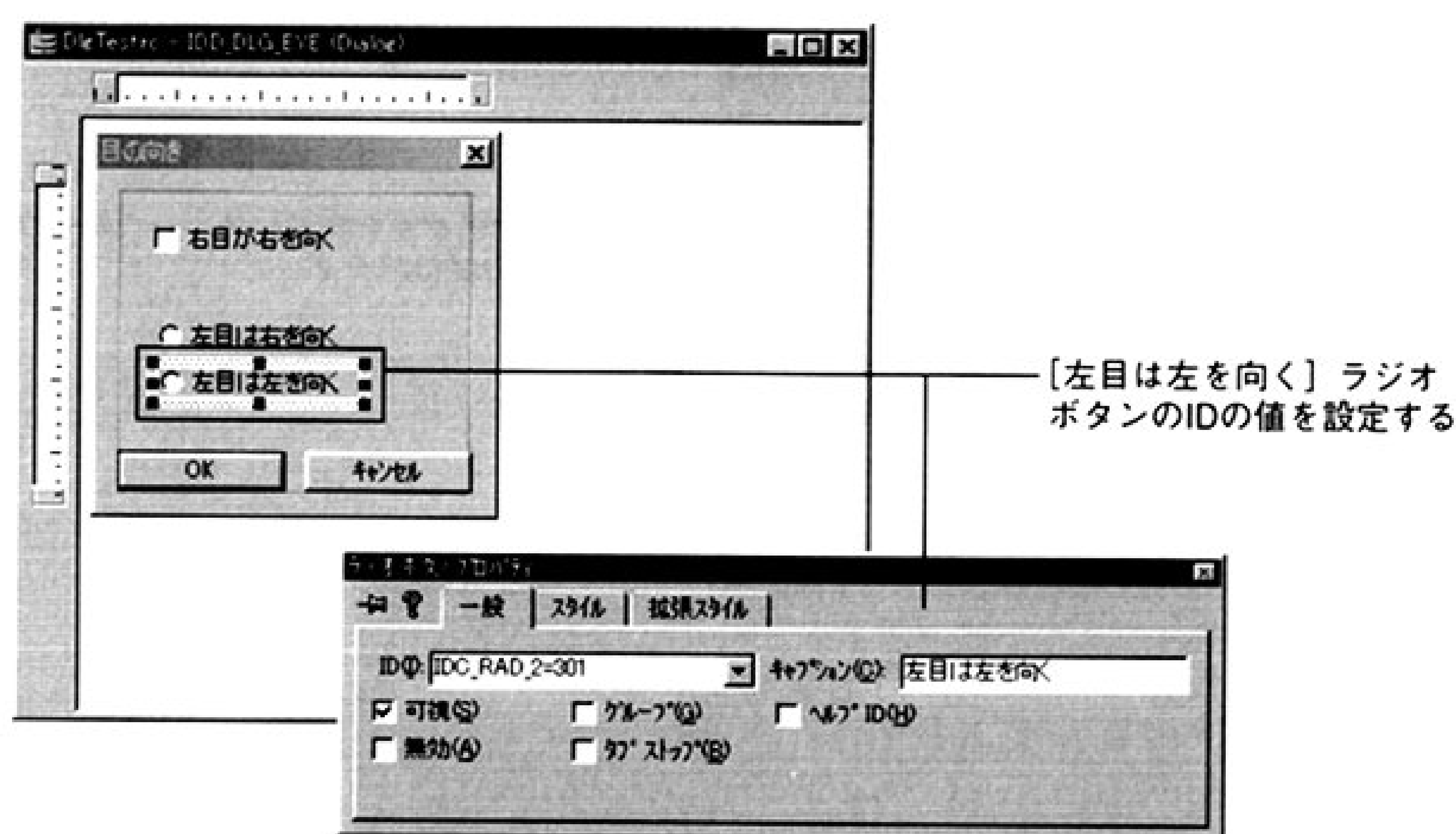


図3-22　［左目は左を向く］ラジオボタン（IDC_RAD_2）のプロパティ

最後にラジオボタンを囲むためのグループボックスを配置します。グループボックスは必須ではありませんが、Windowsの慣習には素直に従った方がいいでしょう。なおグループボックスはメッセージをいっさい発生しません。このようなメッセージを発生しないコントロールには、「IDC_STATIC」という特別なIDが与えられます(**図3-23**)。

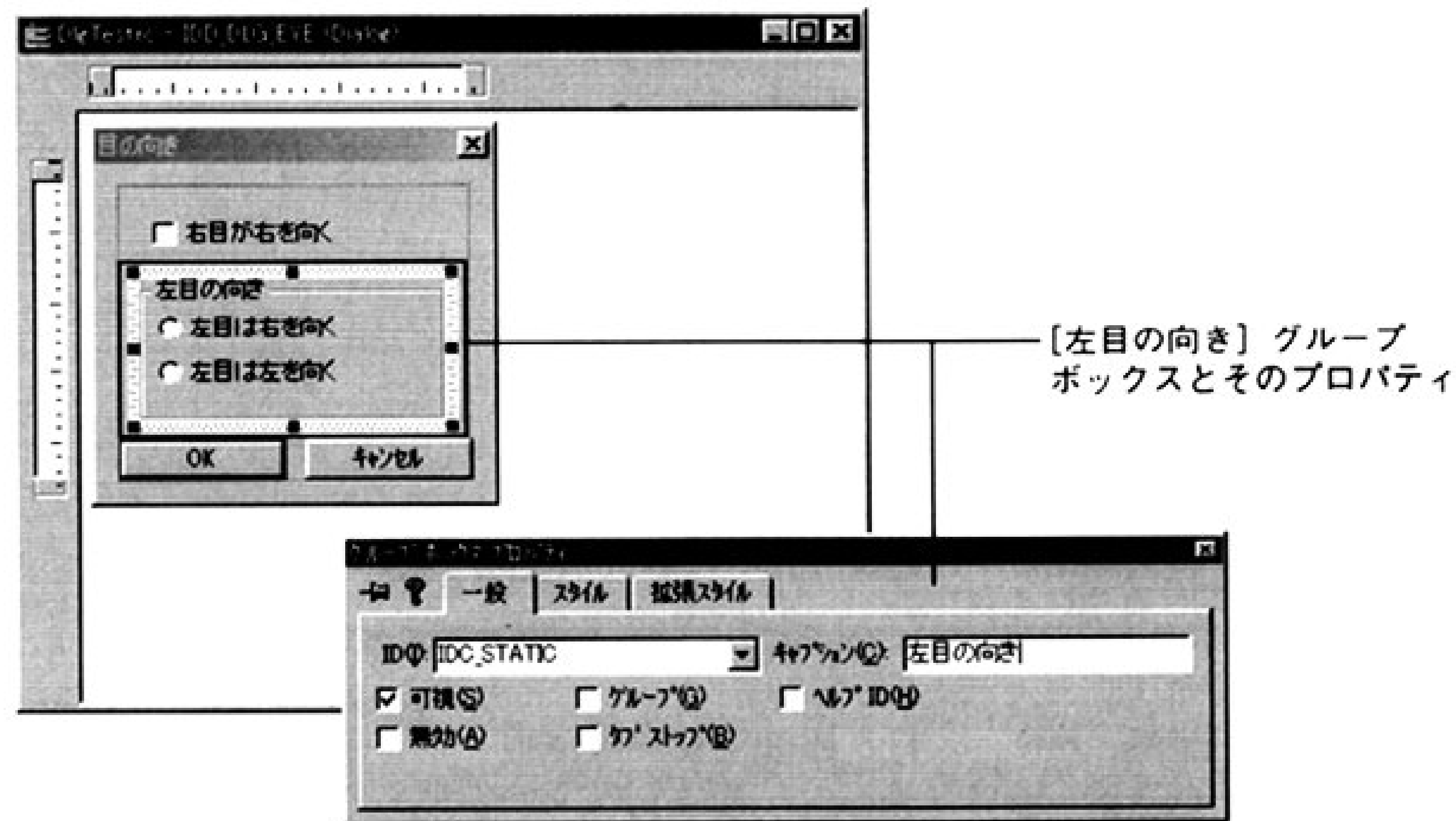


図3-23　グループボックス「左目の向き」のプロパティ

また、**タブオーダー**の設定にも気を配る必要があります。Ctrl+Dを押すと(あるいは[レイアウト]－[タブオーダー]コマンドを実行すると)、各コントロールの脇に現在のタブオーダーが数字で示されます。並べ直したい順にコントロールをクリックしてください。最初にクリックしたコントロールのタブオーダーが1、次にクリックしたものが2、と変わっていきます。最終的に**図3-24**のように設定し、再びCtrl+Dを押せば、そこでタブオーダーの変更は完了します。

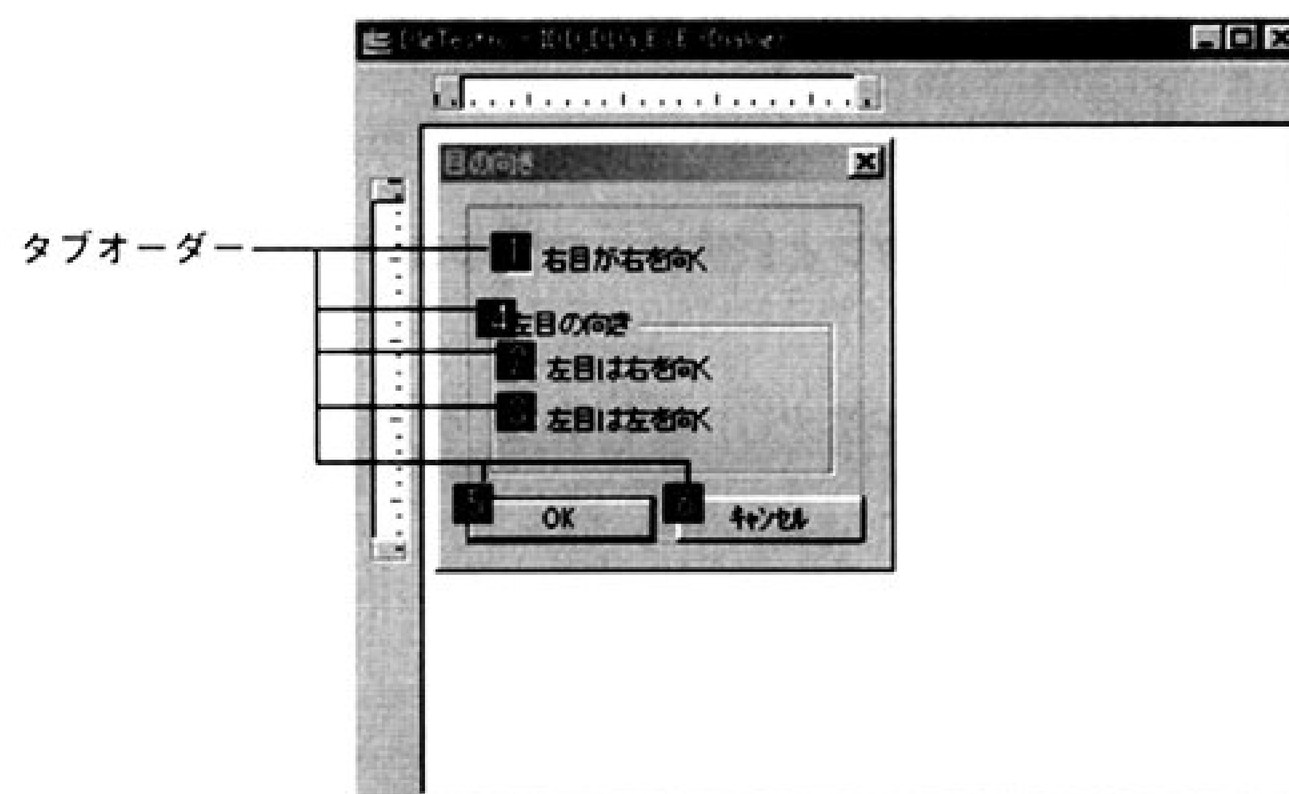


図3-24　タブオーダーの設定

さらにタブオーダーの順で**グループの先頭にあたるラジオボタンと、グループの最後のラジオボタンの1つ先のコントロール**は、プロパティボックスの［グループ］をチェックしなければなりません。今、作成しているダイアログボックスでは、ラジオボタンのタブオーダーは2から3までなので、**図3-25**のようにタブオーダーが2のラジオボタンと、4のグループボックスについて［グループ］チェックボックスをチェックするわけです。

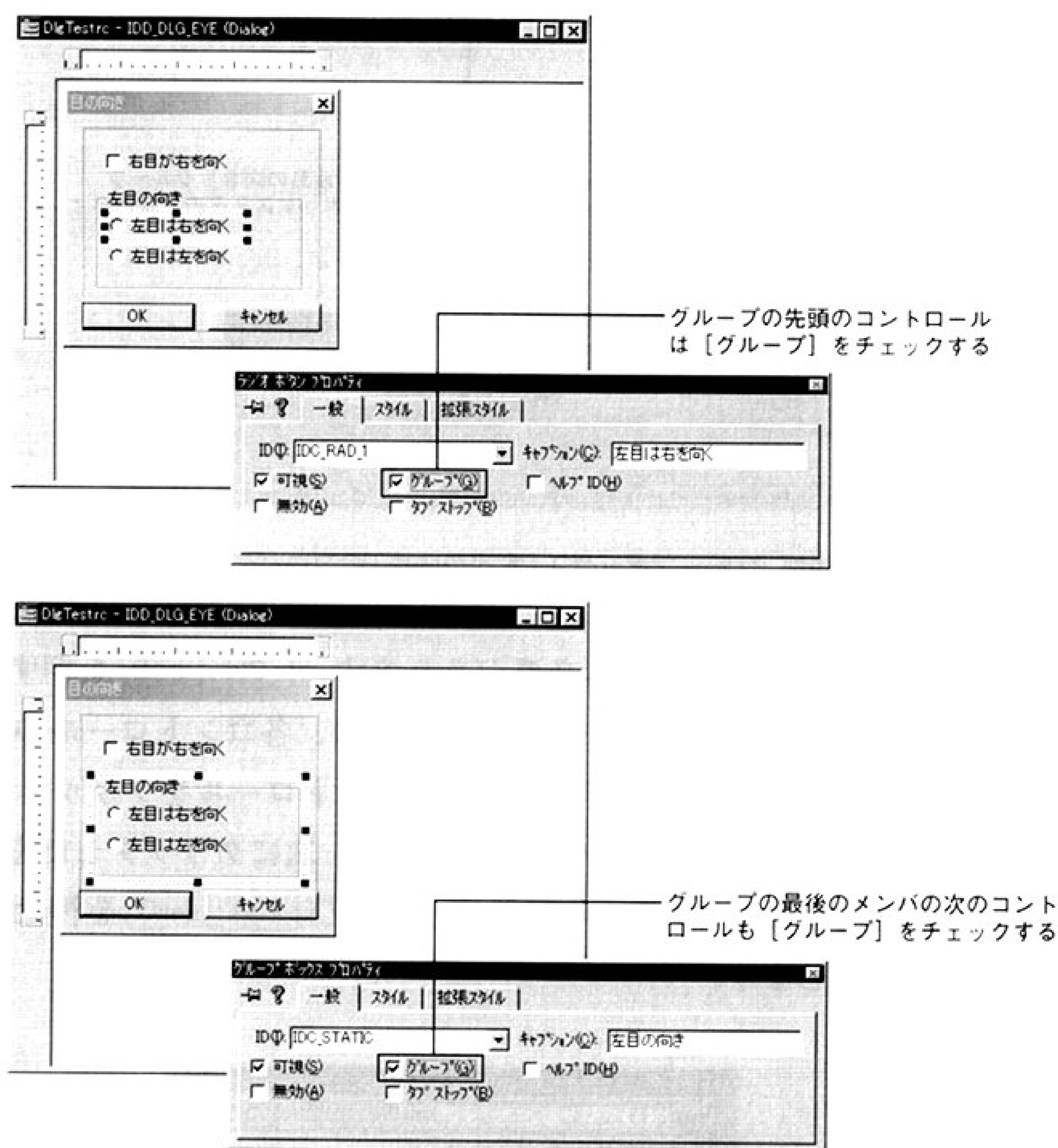


図3-25　先頭のラジオボタンと最後の1つ先のコントロールのプロパティ

以上でダイアログボックスの作成が終わりました。クラスを登録するため、ClassWizardを起動し、［クラスの新規作成］ダイアログボックスで［クラス名］に［CEyeDlg］を指定してください（**図3-26**）。

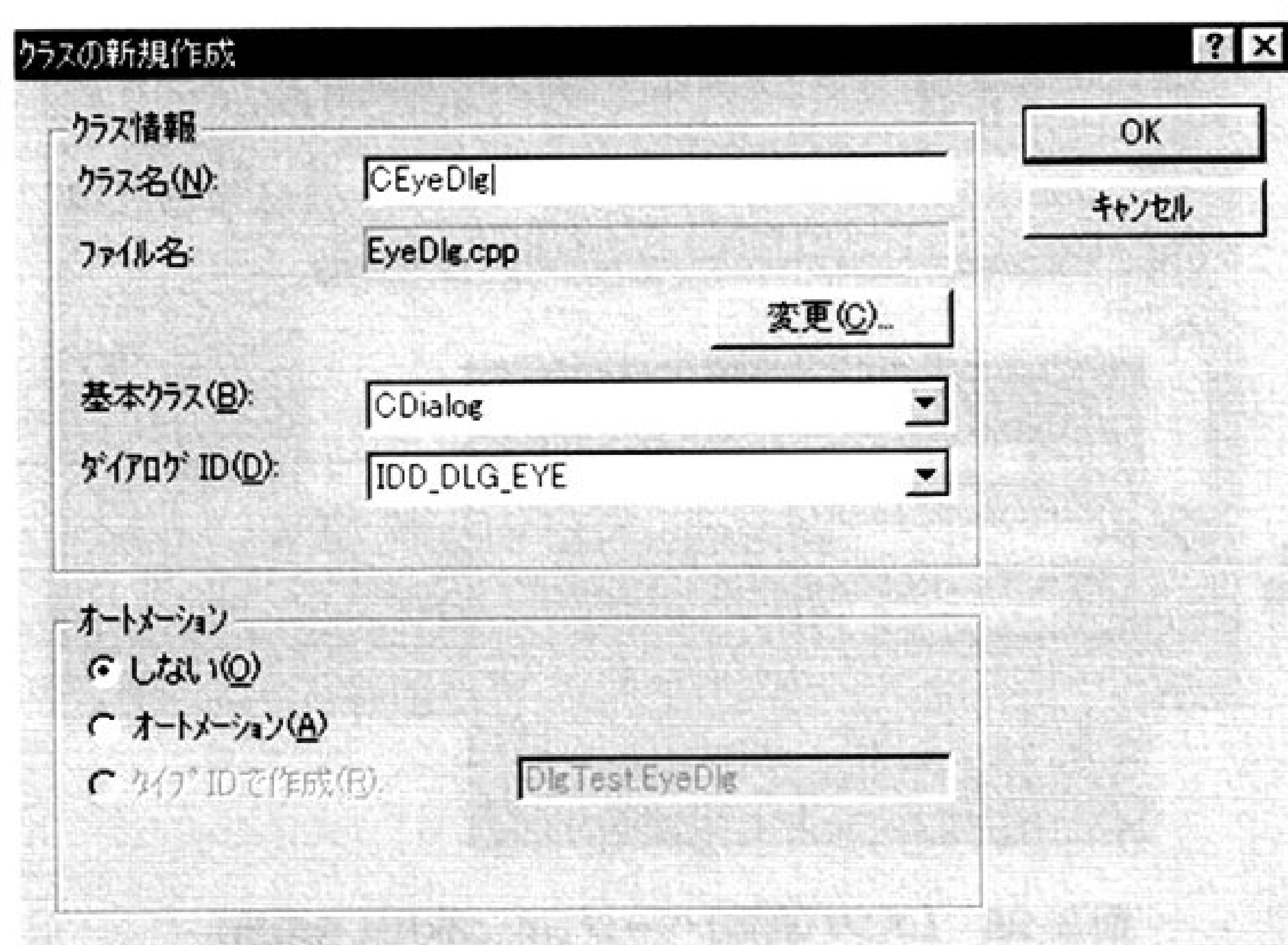


図 3-26 [クラスの新規作成] ダイアログボックスで [CEyeDlg] というクラス名を指定

さて、ここからがDDXのための設定です。ClassWizardの[メンバ変数] タブをクリックしてください。ClassWizardに[メンバ変数] ページが表示され、ダイアログボックスに配置したコントロール一覧が表示されます(**図 3-27**)。

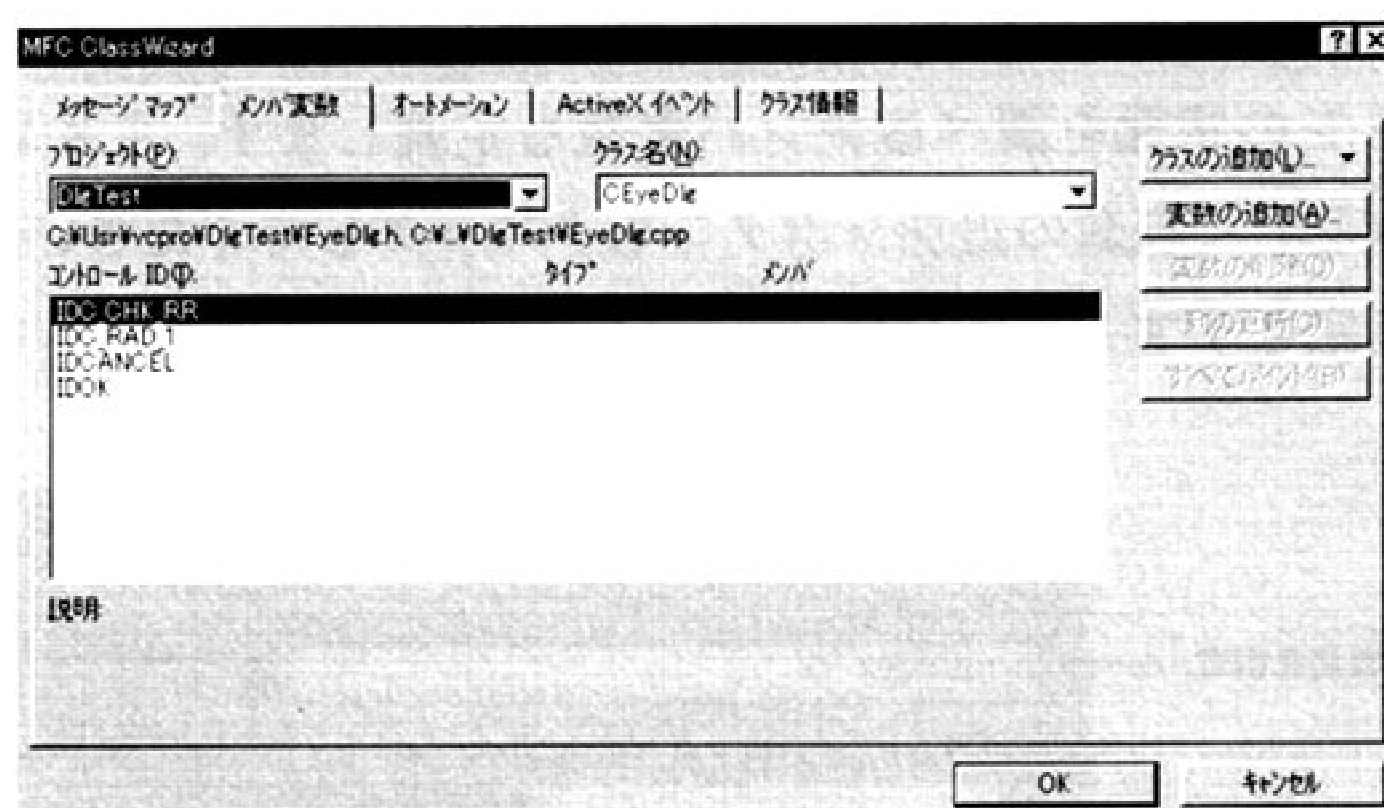


図 3-27 [メンバ変数] ページ

一覧リストに表示されるのは、基本的にはDDXを利用可能なすべてのコントロールです。ただしグループにまとめられているラジオボタンに関しては、グループの先頭の(タブオーダーが一番小さい)ラジオボタンだけが表示されます。

それではまず、[右目が右を向く] チェックボックスに対応するメンバ変数を追加しましょう。IDC_CHK_RRを選択し、<変数の追加>ボタンをクリックしてください。そして、[メンバ変数の追加] ダイアログボックスで[メンバ変数] に変数名を指定します。ここでは[m_chkRR] としておきましょう(**図 3-28**)。

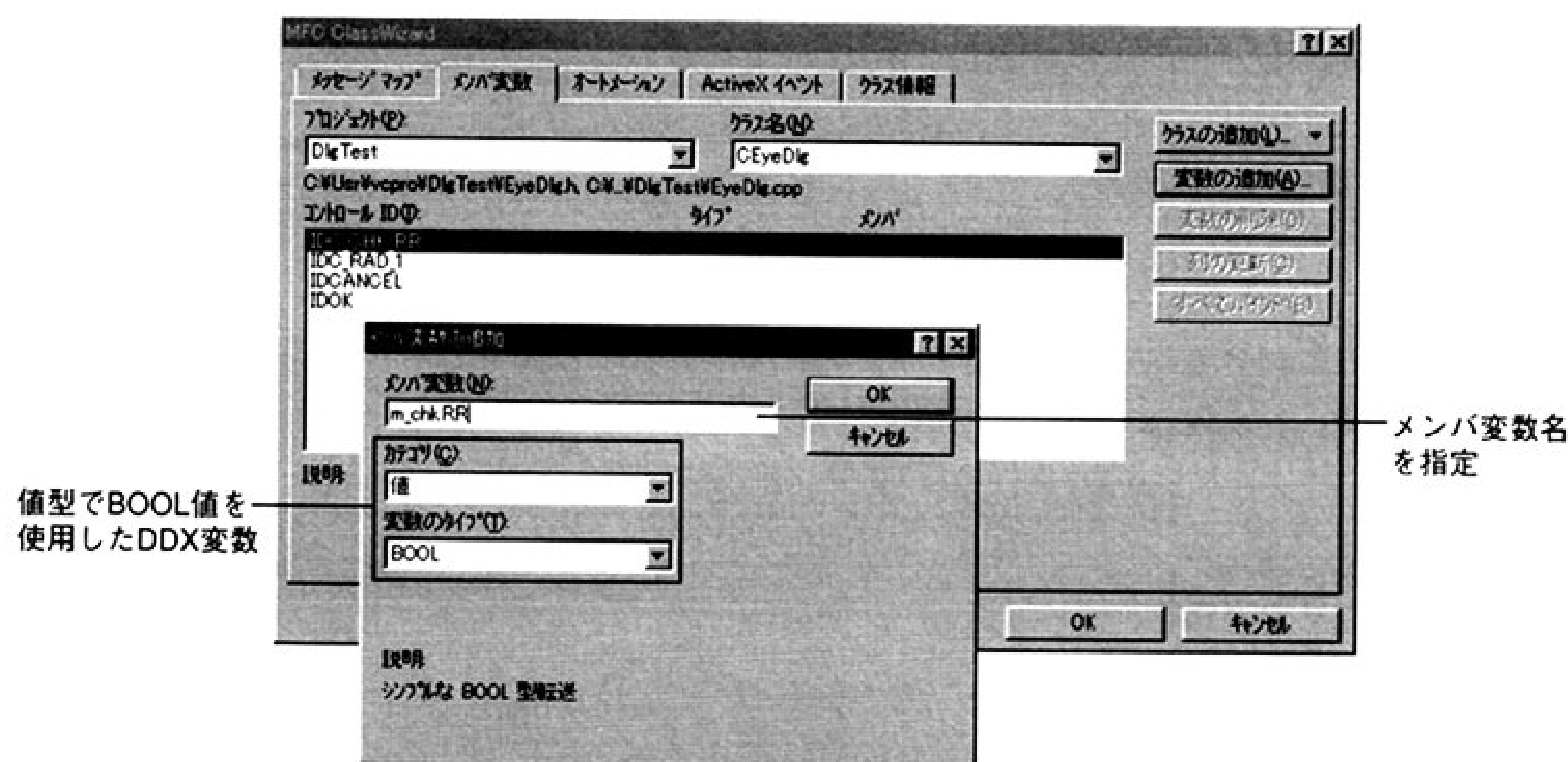


図 3-28 [メンバ変数] ページ (m_chkRR を追加)

このとき [カテゴリ] ボックスに [値] と表示されていることに注意してください。これが値型の DDX であることを示します。なお [カテゴリ] に [コントロール] を選択すれば、コントロール型の DDX 変数も作成できます。

また [変数のタイプ] が [BOOL] と表示されていることからわかるように、この変数はチェックボックスのチェックの有無に応じて、TRUE または FALSE という値を取ります。一般にチェックボックスに対する値型の DDX 変数は BOOL 型になります。

続いてラジオボタンの状態を調べるメンバ変数も定義します。[メンバ変数] ページで IDC_RAD_1 を選択し、<変数の追加>ボタンをクリックしてください。メンバ変数名は、「m_radIndex」と指定します (**図 3-29**)。

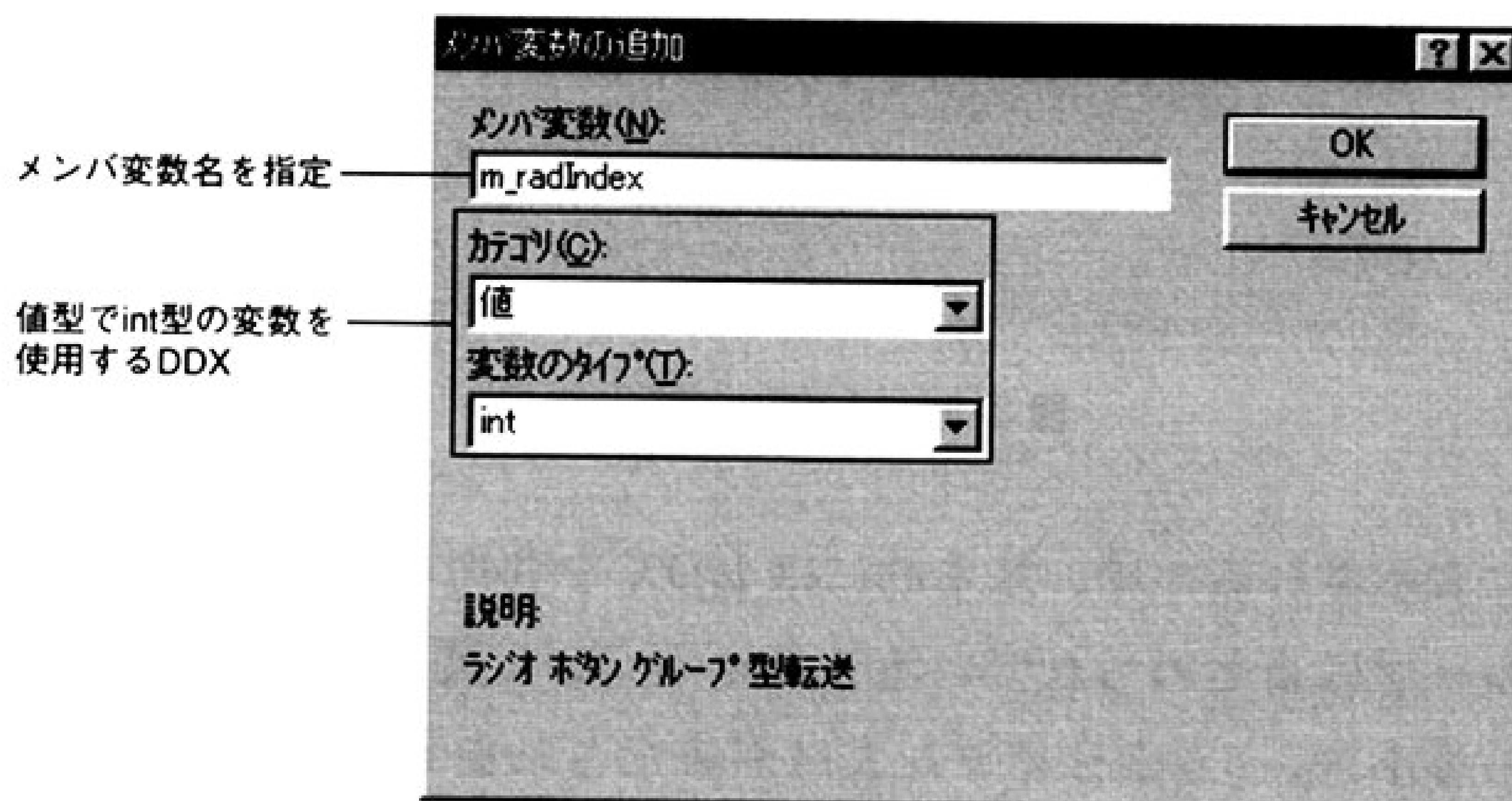


図 3-29 [メンバ変数] ページ (m_radIndex を追加)

ラジオボタンに対する値型の DDX 変数は、先頭のラジオボタンが選択されている場合に 0、その次ならば 1……、という連続した整数値を取ります。

変数の追加が終了したら、<OK>ボタンをクリックして、ClassWizard を終了します。次に、これらの DDX を実際にプログラムの中で利用してみましょう。まず、ダイアログボックスを［目の向き］メニューで表示することにします。**表 3-7** のような設定でメニューのメッセージハンドラを作成してください。

クラス名	CDlgTestView
オブジェクト ID	ID_MENU_EYE
メッセージ	COMMAND
関数名	OnMenuEye

表 3-7　DDX を利用したダイアログボックスを表示するメッセージハンドラの指定

出力されたスケルトンを**リスト 3-10** のようなプログラムに変更します。

リスト 3-10　CDlgTestView::OnMenuEye 関数（DlgTestView.cpp）

```
#include "EyeDlg.h"

void CDlgTestView::OnMenuEye()
{
  CEyeDlg dlg;

  dlg.m_chkRR = (RtEyePos < 0 ? TRUE : FALSE);      // m_chkRR の設定
  dlg.m_radIndex = (LtEyePos < 0 ? 0 : 1);          // m_radIndex の設定
  if (dlg.DoModal() == IDOK) {
    RtEyePos = (dlg.m_chkRR ? -10 : 10);
    LtEyePos = (dlg.m_radIndex == 0 ? -10 : 10);
    InvalidateRect(NULL, FALSE);
  }
}
```

このプログラムではまず CEyeDlg クラスのオブジェクト dlg を作り、続いて dlg.m_chkRR および dlg.m_radIndex という 2 つの DDX 変数が現在の顔の表情を反映するように設定します。dlg.m_chkRR は、右目が右を向いている（RtEyePos<0）ときは TRUE、右目が左を向いていれば FALSE です。dlg.m_radIndex はラジオボタンの順番ですから、左目が右を向いている（LtEyePos<0）ならば 0 を、左目が左を向いていれば 1 に設定します。

ここで DoModal 関数を実行すると、その中で CDialog::OnInitDialog 関数が呼び出され、さらにその中から CWnd::UpdateData 関数が呼び出されます。UpdateData 関数はダイアログボックスクラスのオブジェクトとダイアログボックスの間でデータ交換を行うもので、次のように FALSE を引数とするとダイアログボックスへデータを転送し、TRUE を指定

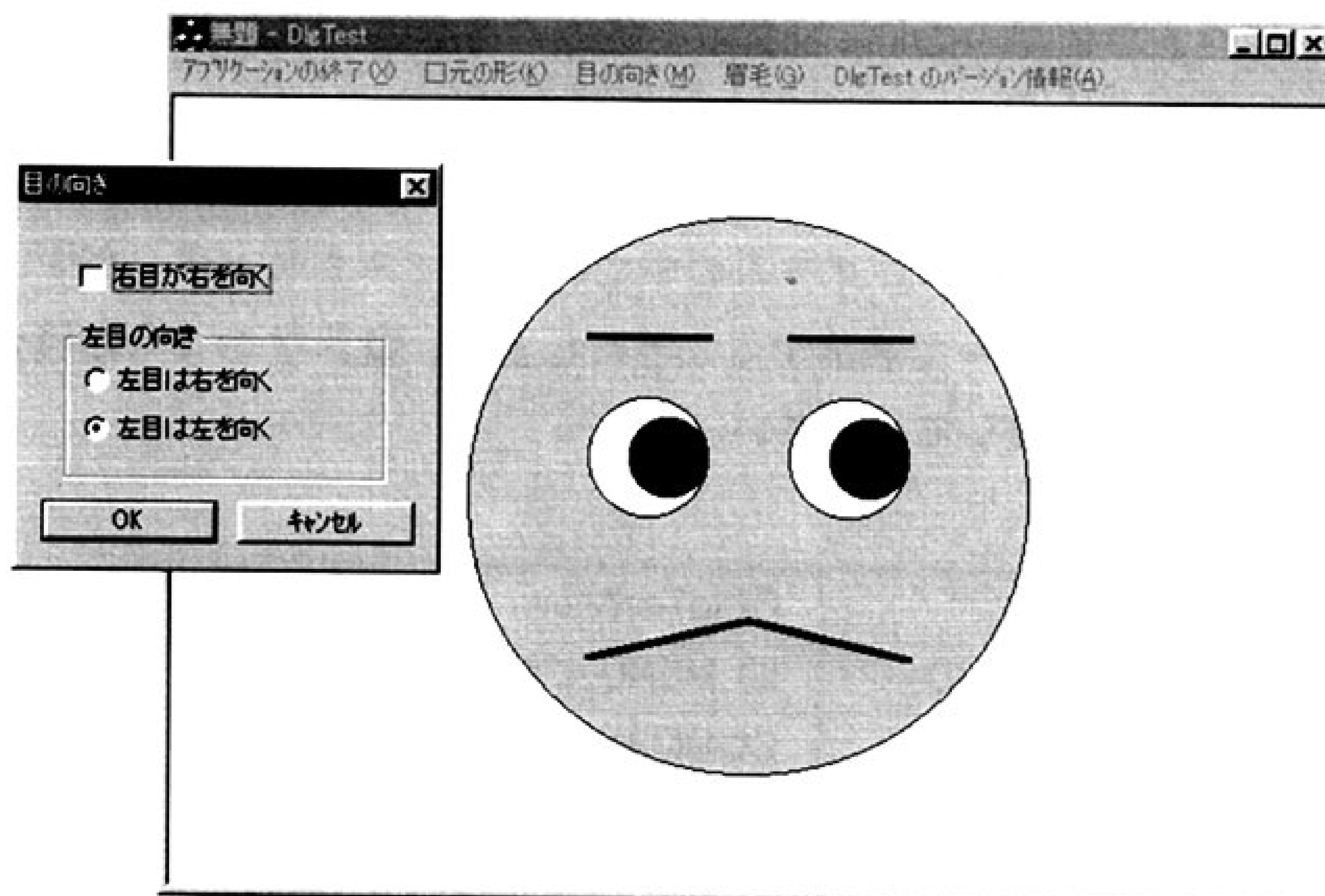


図 3-30 実行画面

するとダイアログボックスからデータを読み取ります。

```
CMyDialog dlg;

dlg.UpdateData(FALSE);      // dlg から → ダイアログボックスへ転送
dlg.UpdateData(TRUE);       // dlg へ ← ダイアログボックスから転送
```

すでに述べたとおり、CDialog::OnInitDialog 関数はダイアログボックスが画面に表示される直前に呼び出される関数です。この関数の中では引数を FALSE として UpdateData 関数を実行しています。そのため画面上に表示されるダイアログボックスには常に DDX 変数のデータが反映されるのです。

<OK>ボタンをクリックすると CDialog::OnOK 関数がデフォルトでは呼び出されますが、この関数の中では引数を TRUE として UpdateData 関数が実行されます。そのため<OK>ボタンを押してダイアログボックスを閉じると、ダイアログボックスのデータが DDX 変数に代入されるのです。よって、DoModal 関数が終了した時点で DDX 変数の値を調べれば、ダイアログボックスに対して行った操作が読み取れるわけです。

3.9 DDVで入力データをチェック

チェックボックスに対する値型の DDX 変数は、かならず FALSE か TRUE の値を取ります。ラジオボタンでも DDX 変数の値は一定の範囲の整数となることが決まっています。したがって、これらのコントロールについては、DDX 変数の値の有効性を調べる必要は

ありません。しかし任意の文字列が入力可能なエディットボックスを扱う場合は、入力データが有効なものかどうかを調べる作業が不可欠です。この項では**DDV（Dialog Data Validation）**という機能を利用して、エディットボックスに入力したデータの有効性をチェックする方法を説明しましょう。

まずエディットボックスの利用例として、**図3-31**のようなダイアログボックスを作成します。このダイアログボックスは、眉毛の位置と太さを、それぞれエディットボックスを利用して入力しようというものです。

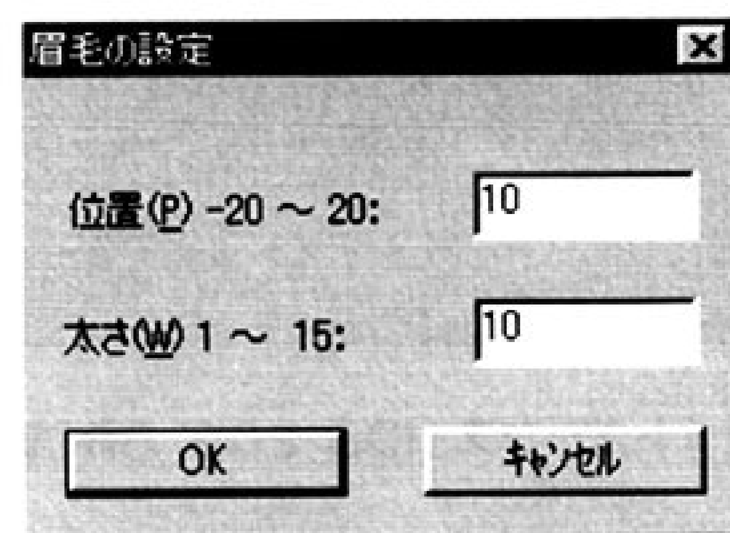


図3-31 ［眉毛の設定］ダイアログボックス

まずリソースエディタでダイアログボックスを新規作成し、<OK>ボタンと<キャンセル>ボタンを**図3-32**の位置に移します。ダイアログボックスのキャプションは［眉毛の設定］、IDは［IDD_DLG_BROW］とします。

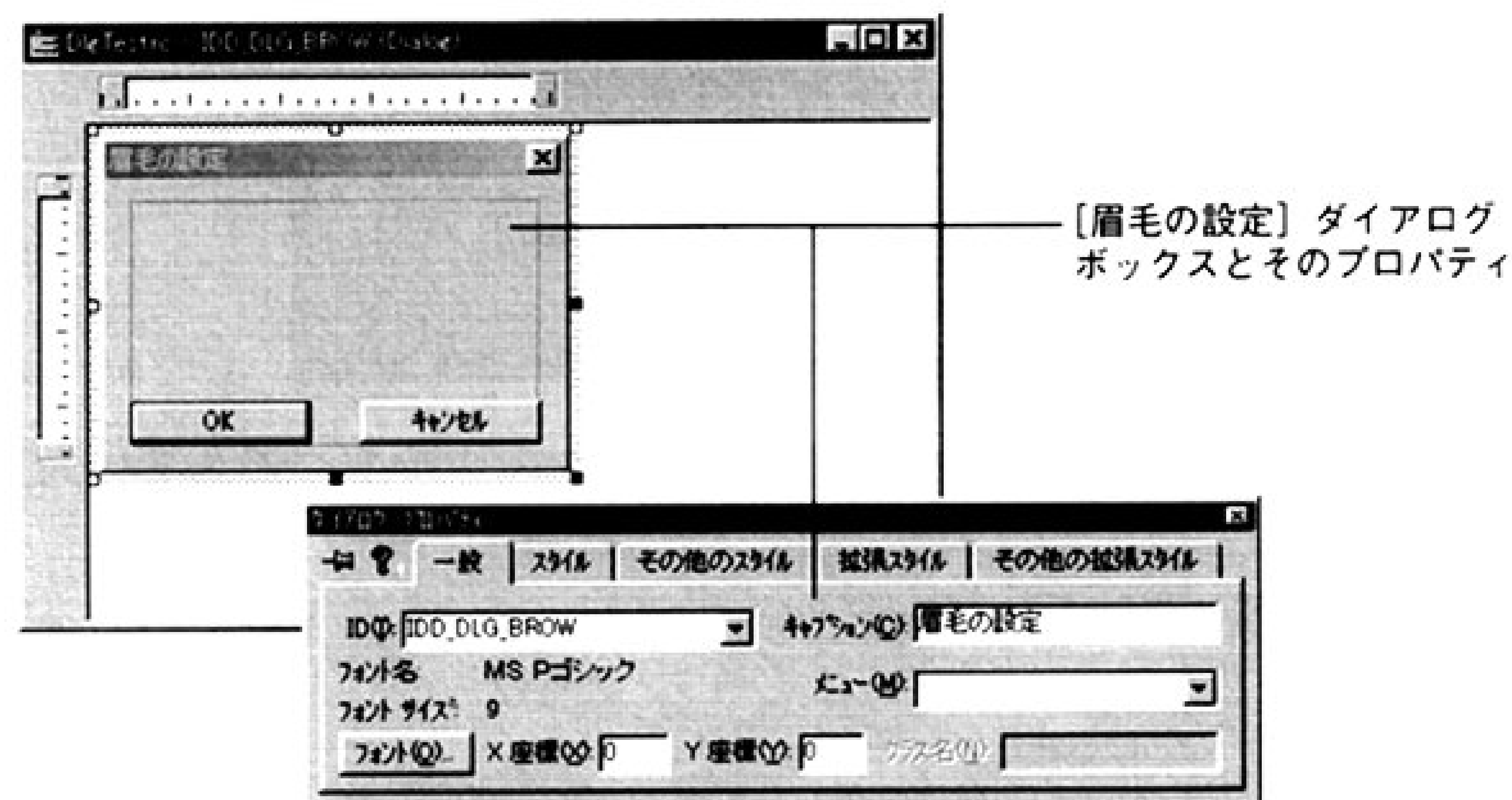


図3-32 ［眉毛の設定］ダイアログボックス（IDD_DLG_BROW）のプロパティ

次にエディットボックスを2つ配置し、それぞれのプロパティを**図3-33**と**図3-34**のように設定します。

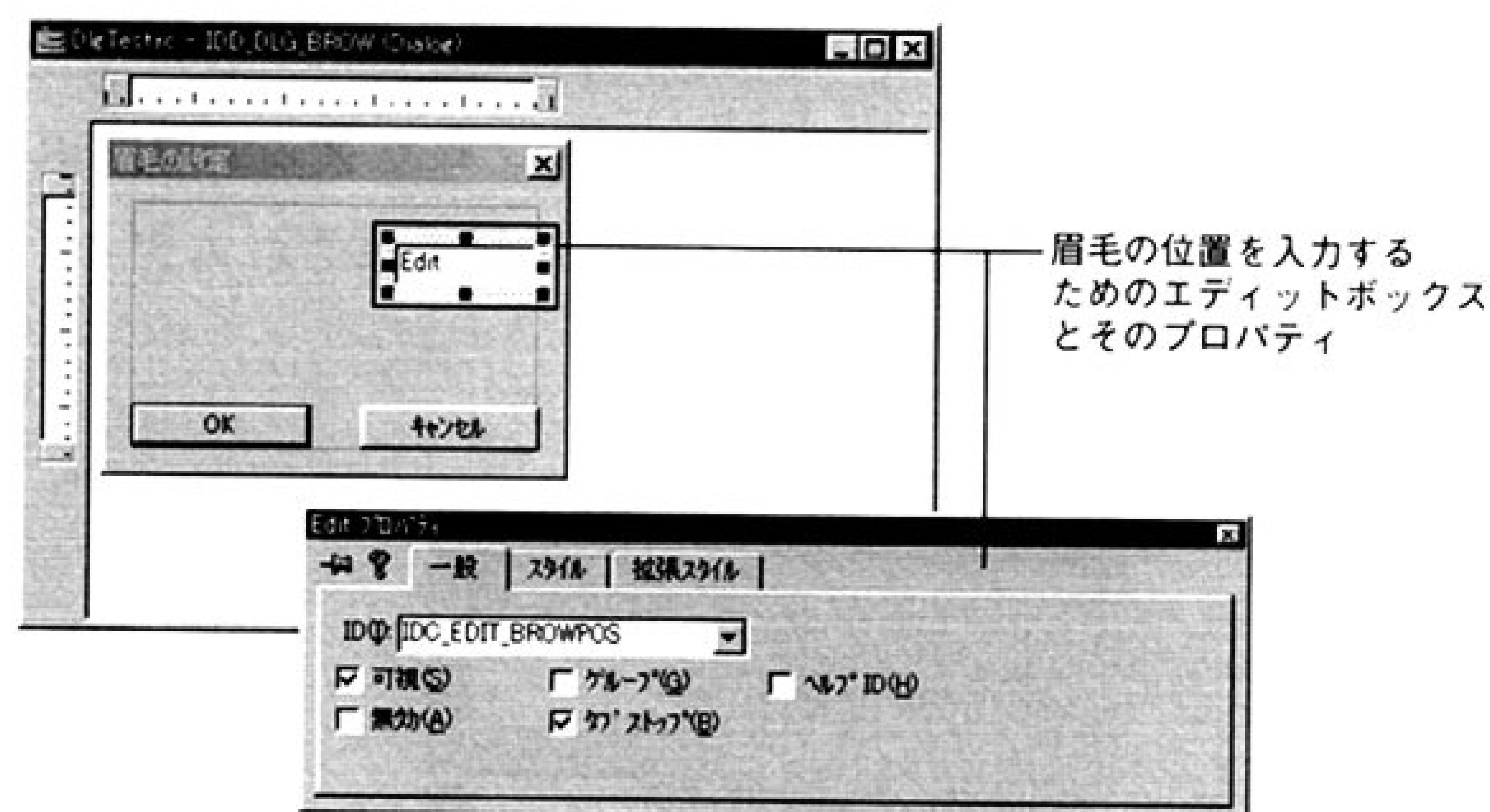


図3-33 位置入力用エディットボックス(IDC_EDIT_BROWPOS)とそのプロパティ

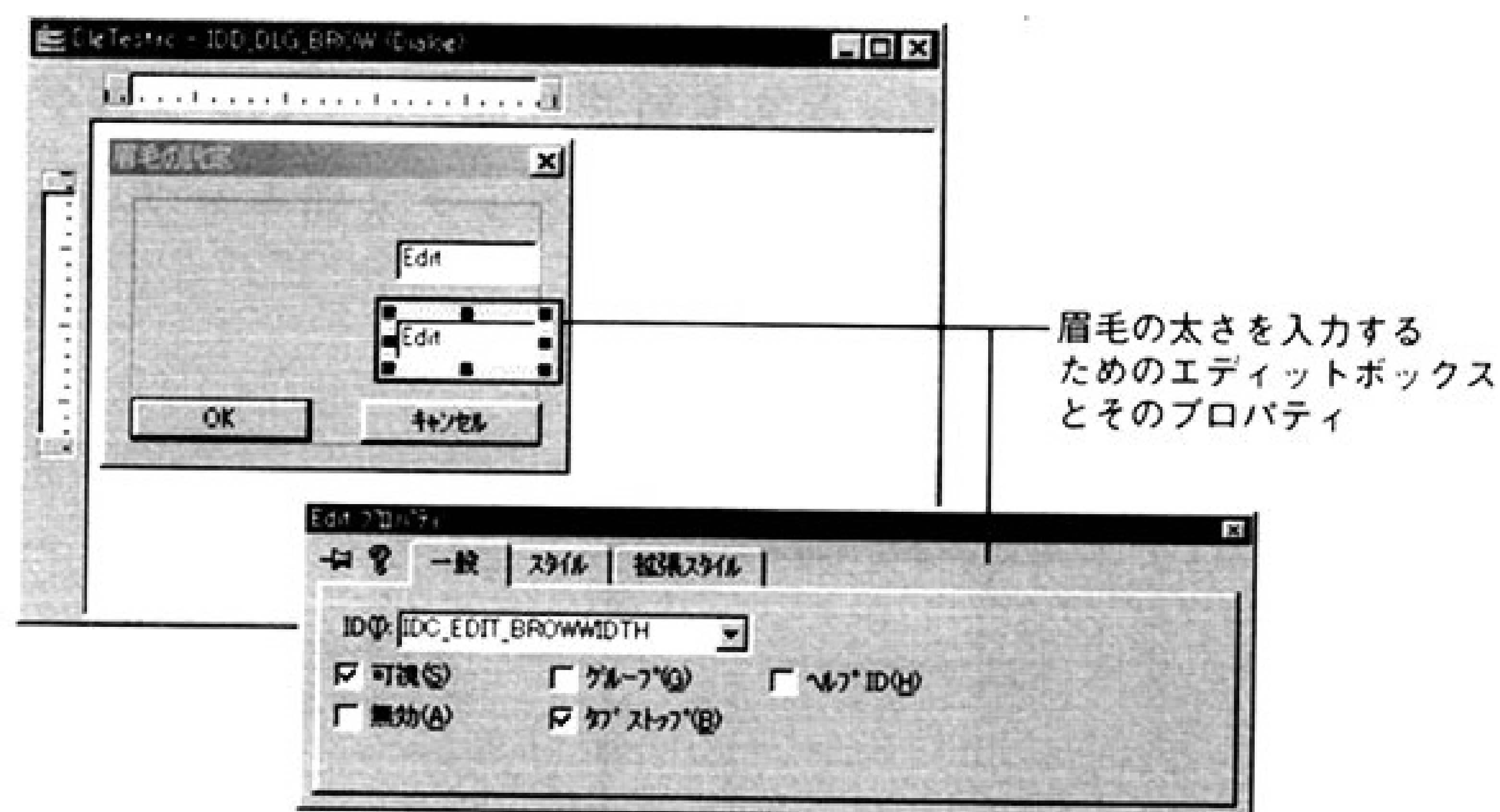


図3-34 太さ入力用エディットボックス(IDC_EDIT_BROWWIDTH)とそのプロパティ

エディットボックスの外見は単なる四角形で、これ自体にはキャプションも何も付きませんので、各エディットボックスの左側にラベルを配置して、エディットボックスの説明を記述することにします（図3-35、図3-36）。

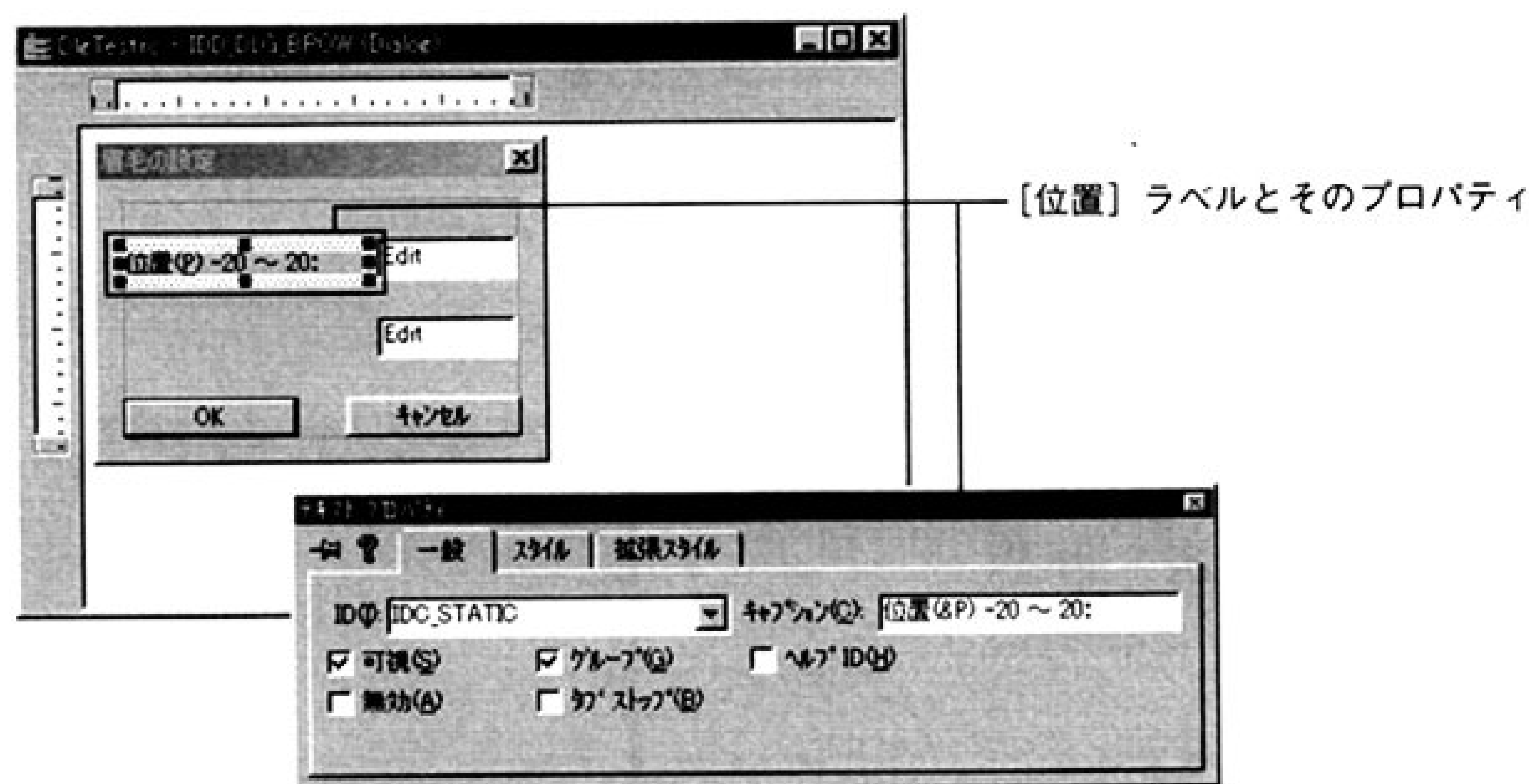


図3-35　[位置] ラベルとそのプロパティ

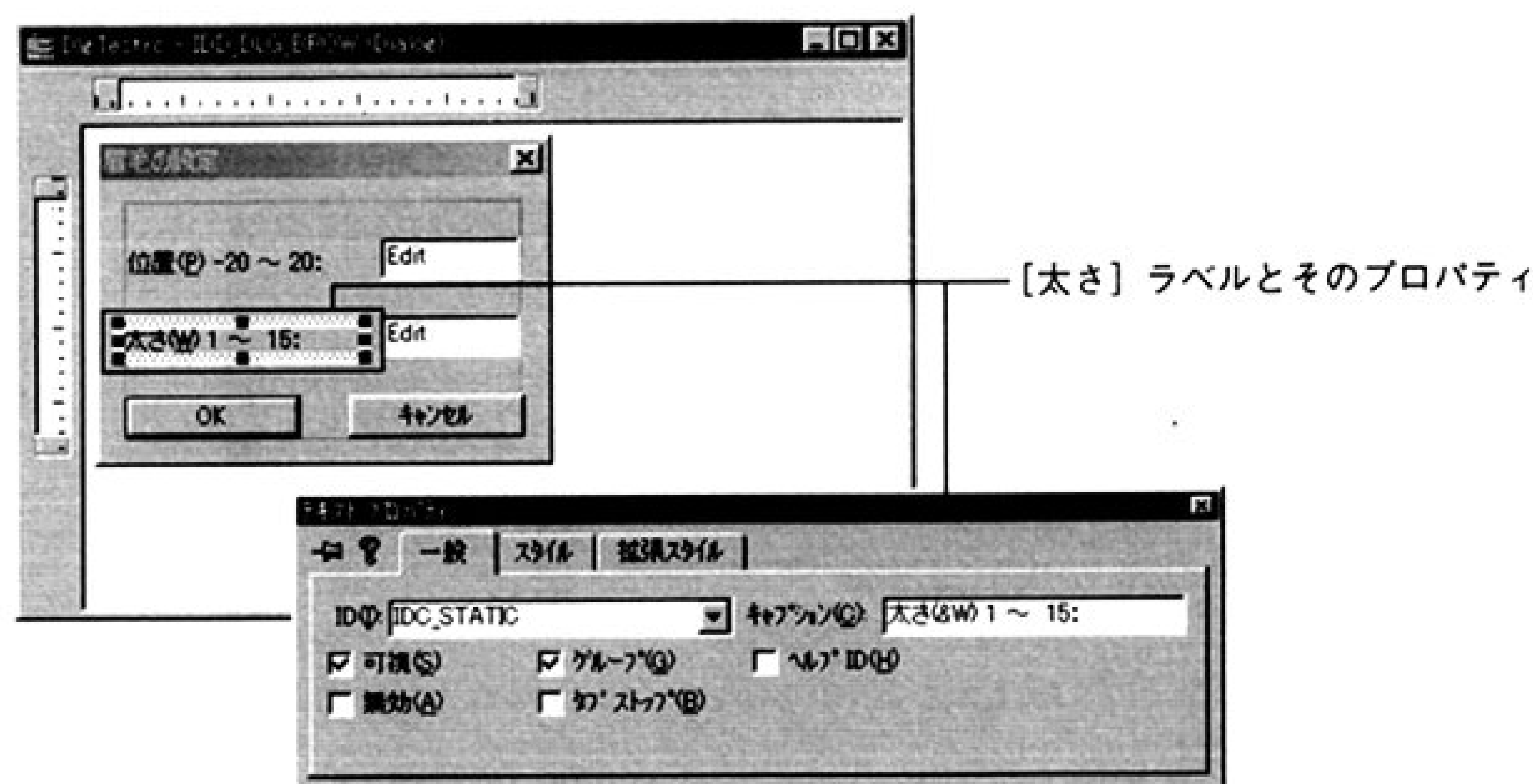


図3-36　[太さ] ラベルとそのプロパティ

このようにエディットボックスにラベルを付けた場合は、**図 3-37** に示すように、対応するラベルとエディットボックスが連続するようにタブオーダーを調整します。こうしておくと、ラベルに指定したアクセスキーでエディットボックスにフォーカスが移動してくれます。たとえばこの例では、Alt + P を押すと位置入力用のエディットボックスが選択され、Alt + W を押すと太さ入力用のエディットボックスが選択されます。

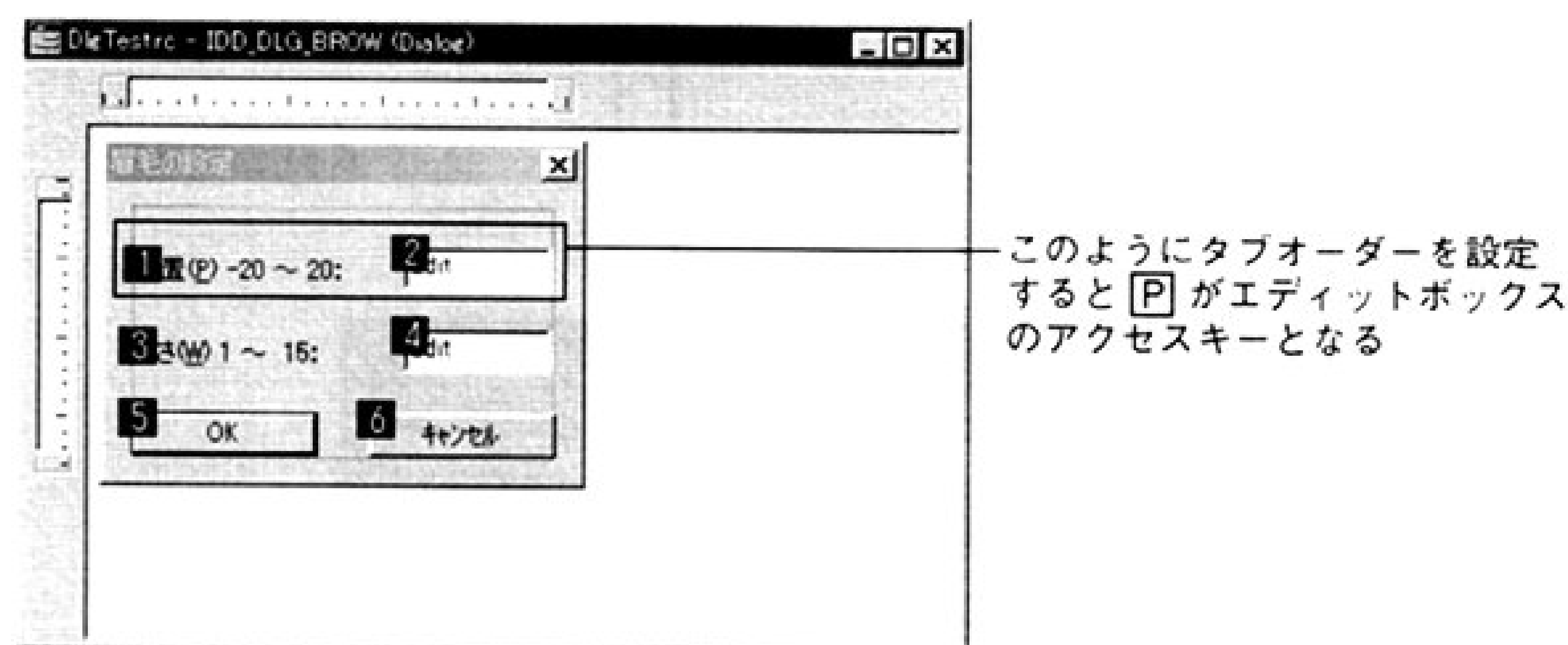


図 3-37 このダイアログボックスのタブオーダー

以上の作業が終了したら、ClassWizard を起動して、このダイアログボックスのクラスをプロジェクトに追加します。クラス名には[CBrowDlg]を指定してください(**図 3-38**)。

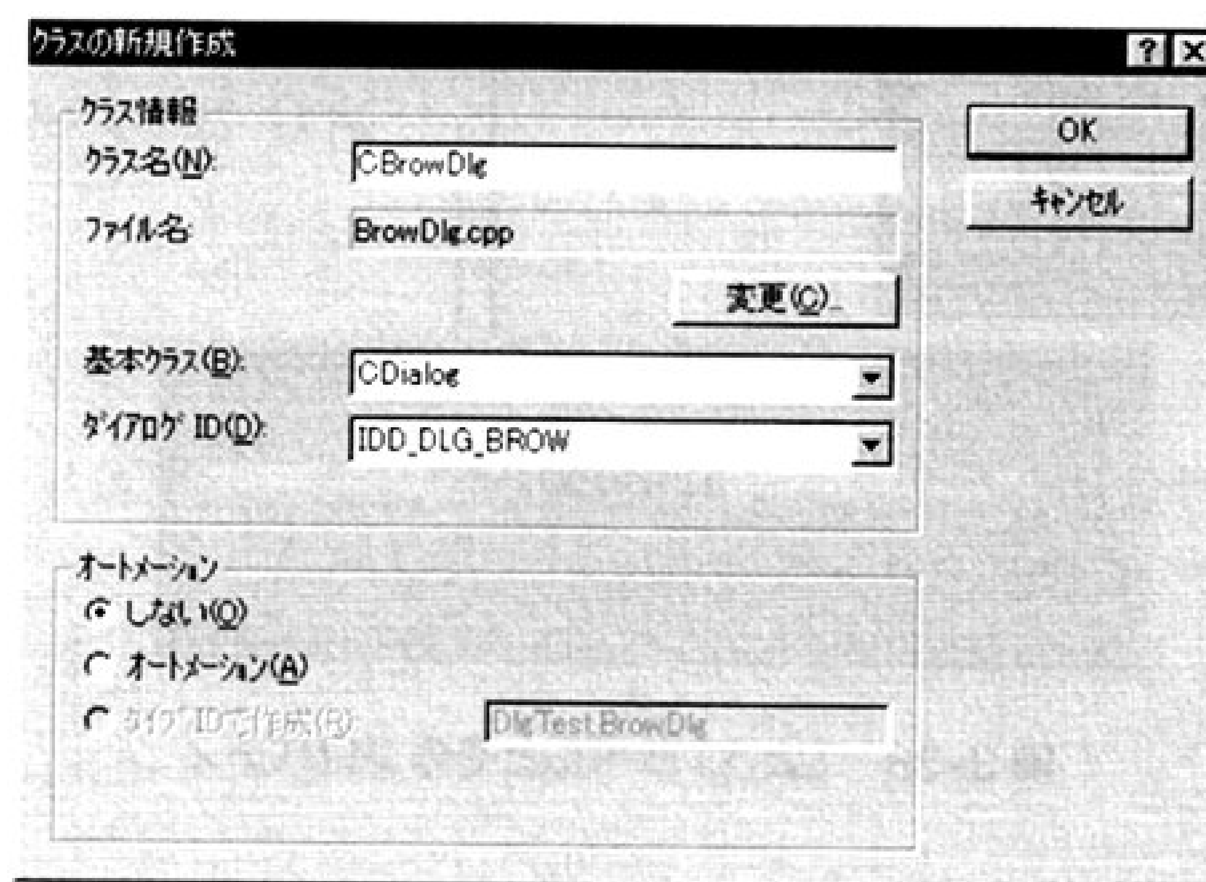
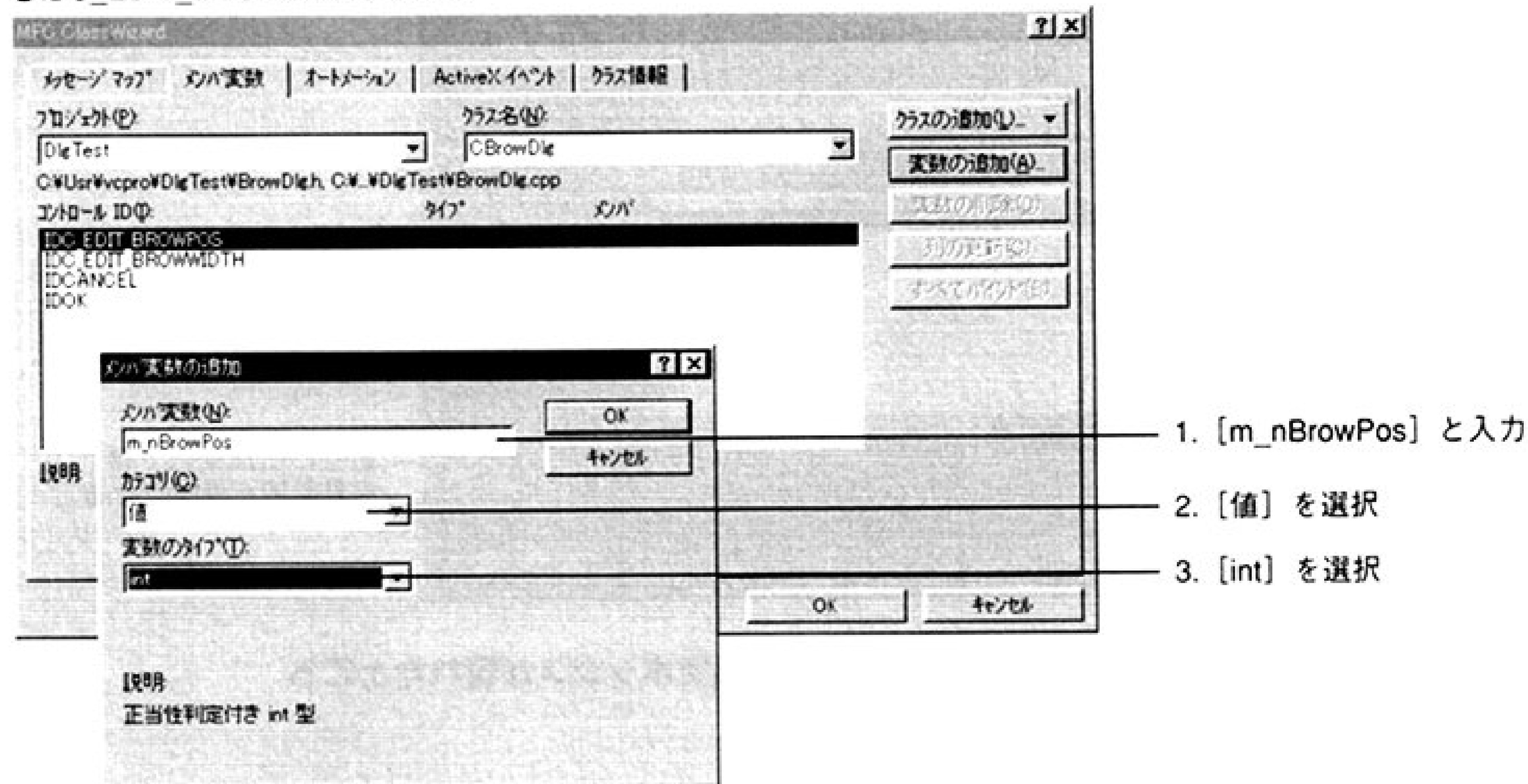


図 3-38 [クラスの新規作成]ダイアログボックスで[CBrowDlg]というクラス名を指定

次にエディットボックス用の DDX 変数を用意します。[メンバ変数]タブをクリックして、[メンバ変数]ページを表示してください。

各エディットボックスの内容を表す値型の DDX 変数として、ここでは**図 3-39** に示すように、「m_nBrowPos(眉毛の位置)」と「m_nBrowWidth(眉毛の太さ)」を作成します。エディットボックス用の値型の DDX 変数には、CString、int、float など、いろいろなデータ型を選ぶことができますが、m_nBrowPos(－20 から 20 までの整数値)には int 型を選びます。m_nBrowWidth(1 から 15 までの整数値)には UINT 型を使ってみましょう。

●IDC_EDIT_BROWPOSについて



●IDC_EDIT_BROWWIDTHについて

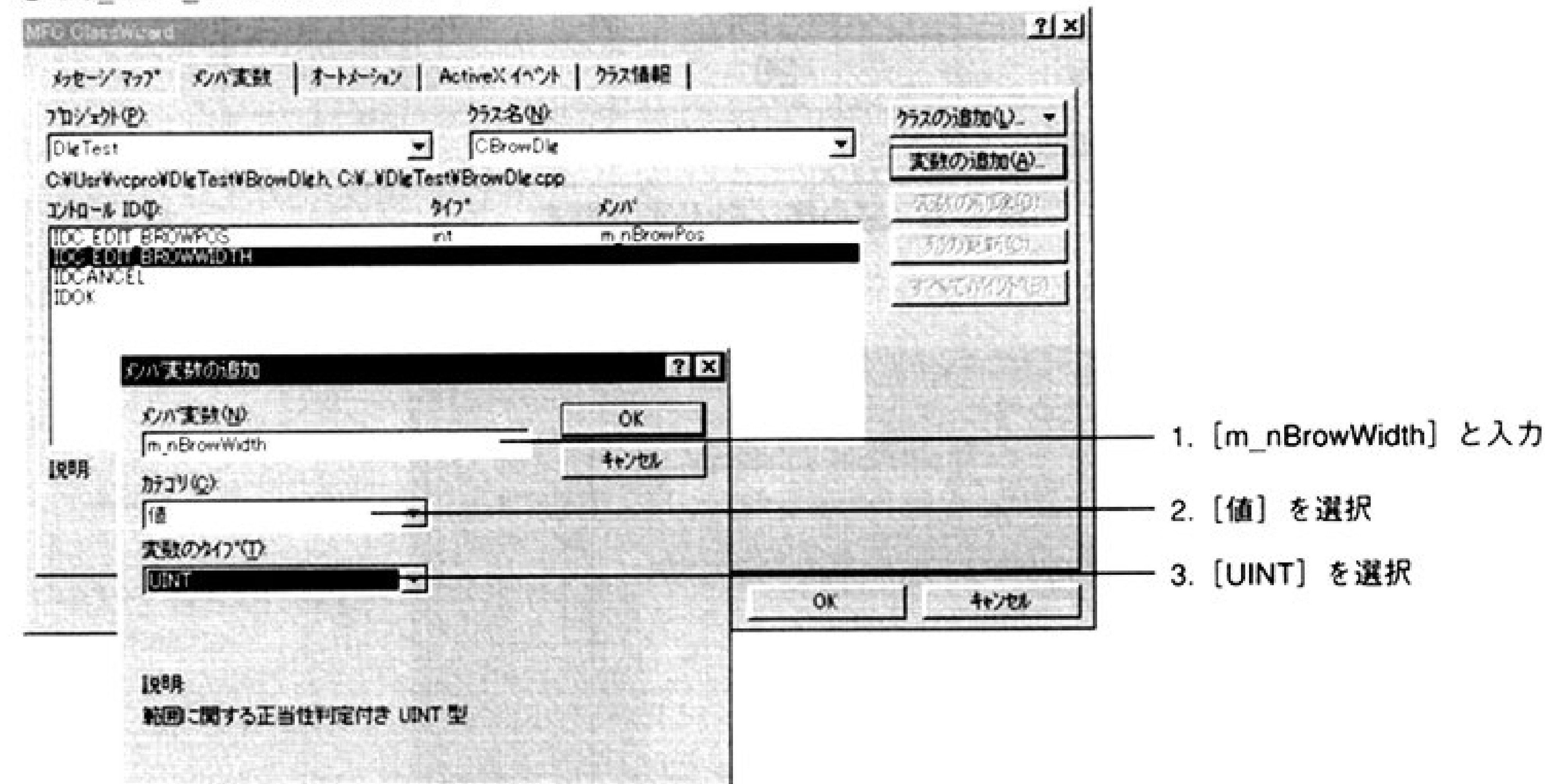


図3-39 [メンバ変数] ページでメンバ変数を作成

エディットボックスに対する値型のDDX変数を作成すると、[メンバ変数の編集] ダイアログボックスの下部で、データの有効範囲の指定ができるようになります。int型やUINT型などの数値形式のDDX変数では、**図3-40**のように [最少値] と [最大値] が指定可能です。なお、今回は使いませんでしたが、CString型のDDX変数の場合は「最大文字数」を指定できます。

ここではそれぞれのDDX変数に、**表3-8**のような最少値と最大値を指定してください（**図3-41**）。

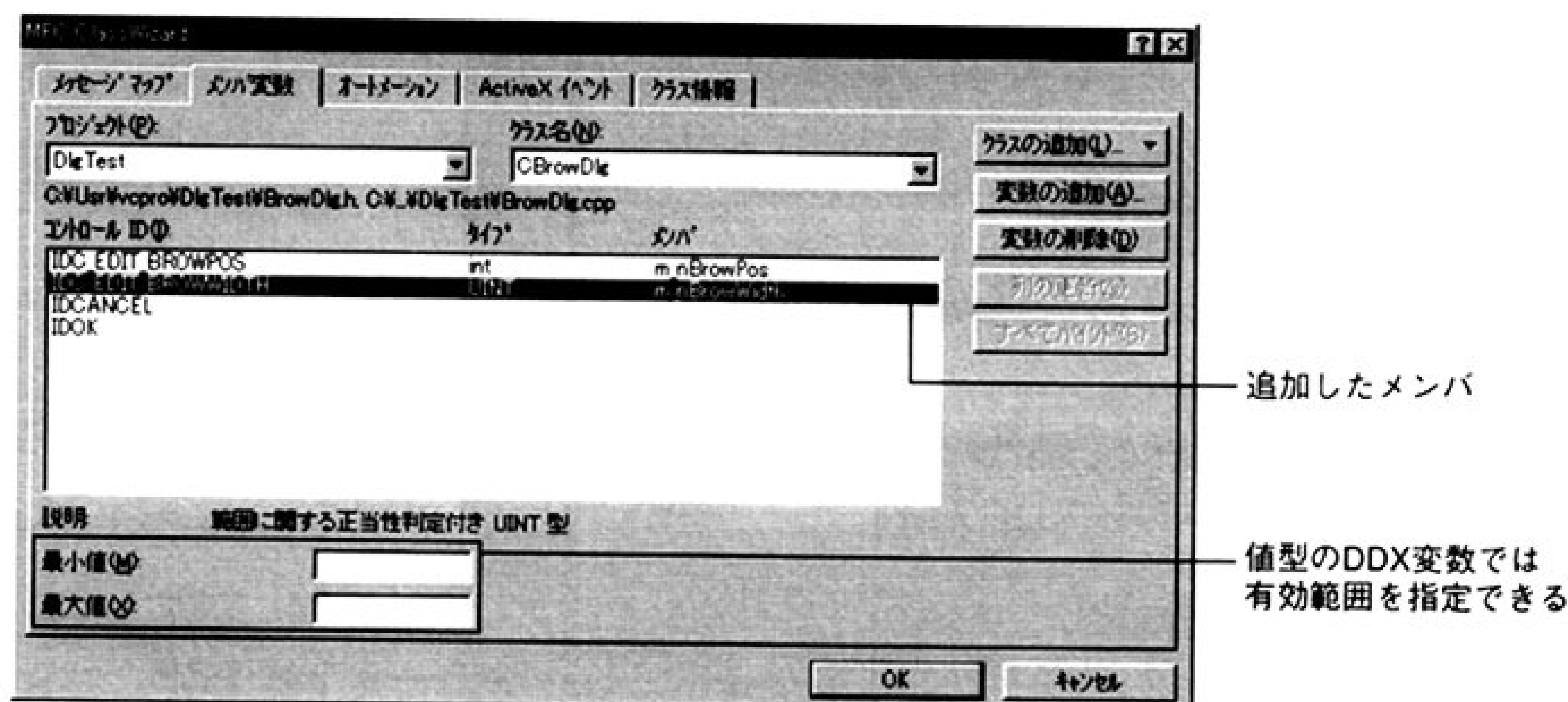


図 3-40　DDX 変数の有効範囲の指定ボックスが現れたところ

メンバ変数	最小値	最大値
m_nBrowPos	-20	20
m_nBrowWidth	1	15

表 3-8　最小値／最大値の指定

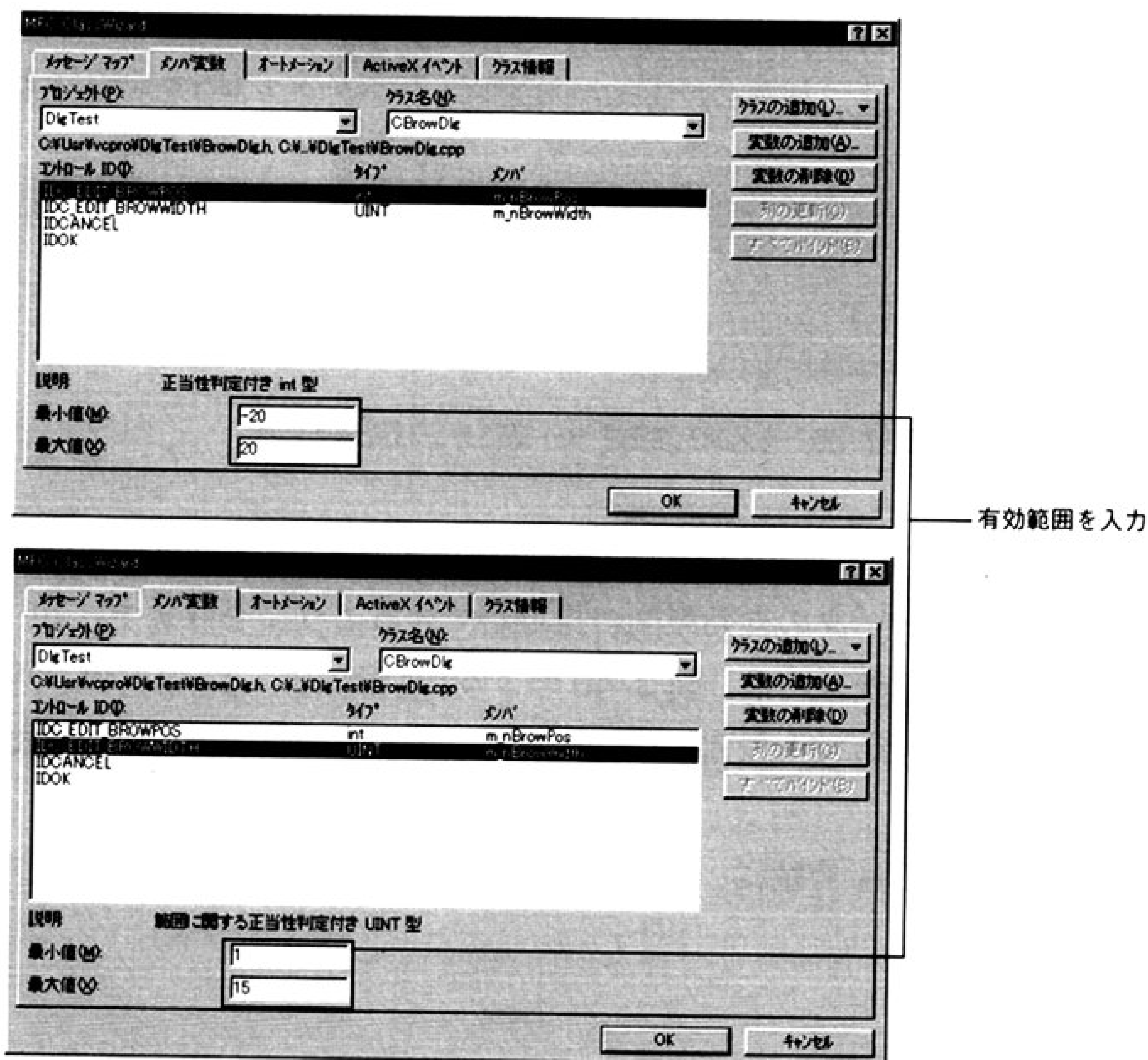


図 3-41　最小値／最大値の設定

以上で[眉毛の設定]ダイアログボックスに関する設定が終わりました。このダイアログボックスは[眉毛]メニューで表示することにします。**表3-9**のような設定でメニューのメッセージハンドラを作成してください。

出力されたスケルトンは**リスト3-11**に示すように修正してください。m_nBrowPosはint型、m_nBrowWidthはUINT型のDDX変数として定義しましたから、BrowPosやBrowWidthなどの整数値が直接代入できます。

クラス名	CDlgTestView
オブジェクトID	ID_MENU_BROW
メッセージ	COMMAND
関数名	OnMenuBrow

表3-9 [眉毛]ダイアログボックスを表示するメッセージハンドラ

リスト3-11 CDlgtestView::OnMenuBrow関数(DLGTEVW.CPP)

```
#include "BrowDlg.h"

void CDlgTestView::OnMenuBrow()
{
  CBrowDlg dlg;

  dlg.m_nBrowPos = BrowPos;
  dlg.m_nBrowWidth = BrowWidth;
  if (dlg.DoModal() == IDOK) {
    BrowPos = dlg.m_nBrowPos;
    BrowWidth = dlg.m_nBrowWidth;
    InvalidateRect(NULL, FALSE);
  }
}
```

それではプログラムをコンパイル/実行してみましょう(**図3-42**)。[眉毛]メニューをクリックすると[眉毛の設定]ダイアログボックスが開きます。まず眉毛の位置のエディットボックスに適当なデータを入力してください。次に[Tab]か[Alt]+[W]を押してフォーカスを移動させ、眉毛の太さのエディットボックスにデータを入力します。ここで<OK>ボタンをクリックすると([Enter]を押してもよい)、ダイアログボックスが閉じ、眉毛の位置と太さが変わります。

ダイアログボックスが期待どおりに動作することを確かめたら、今度はわざと有効範囲外のデータを入力してみましょう。たとえば眉毛の位置を[100]として<OK>ボタンを押してください。すると**図3-43**のようなメッセージボックスが表示され、正しいデータを

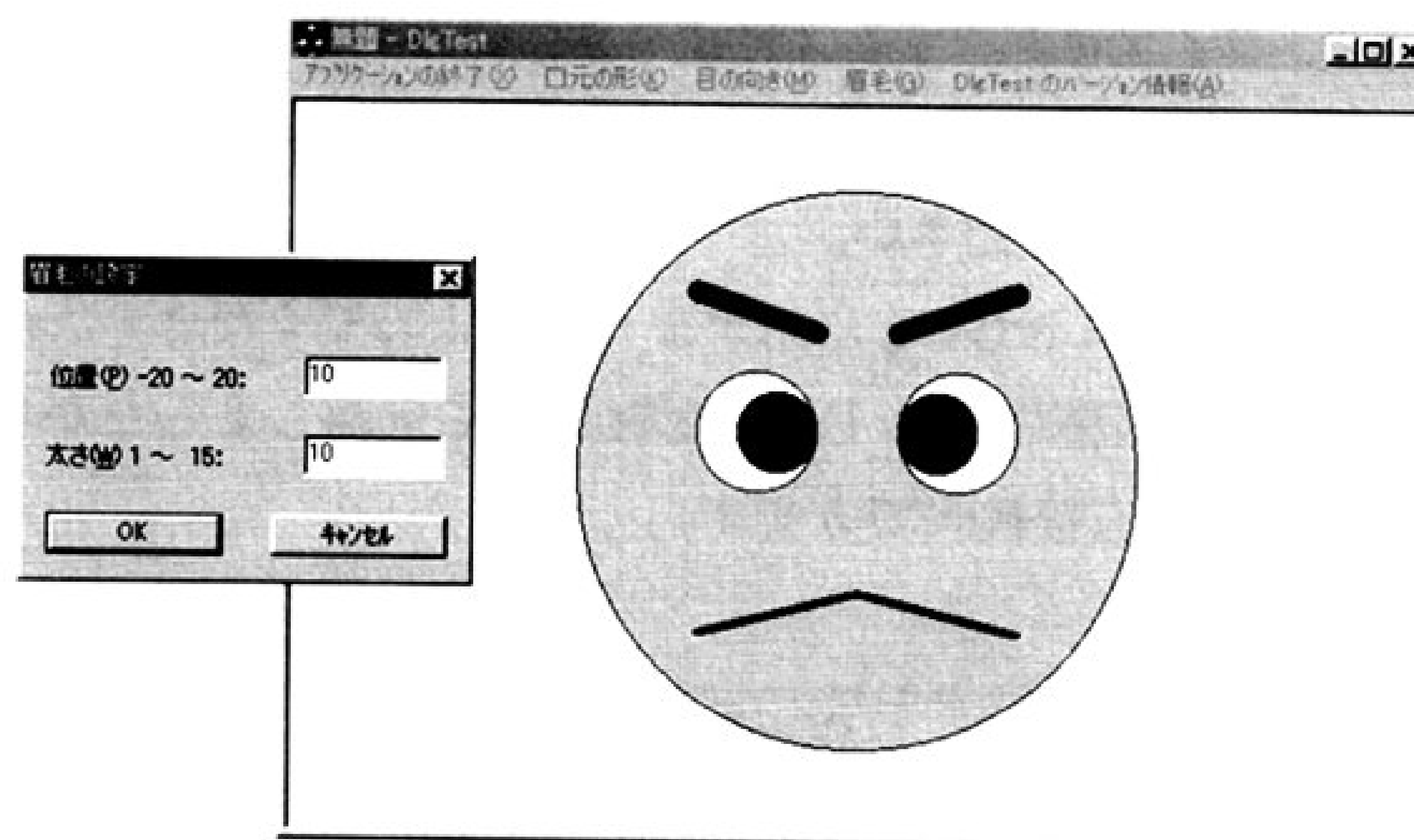


図 3-42　実行画面

再入力するように促されます。自動的にデータの有効範囲チェックが行われたのです。このデータチェック機構が DDV です。

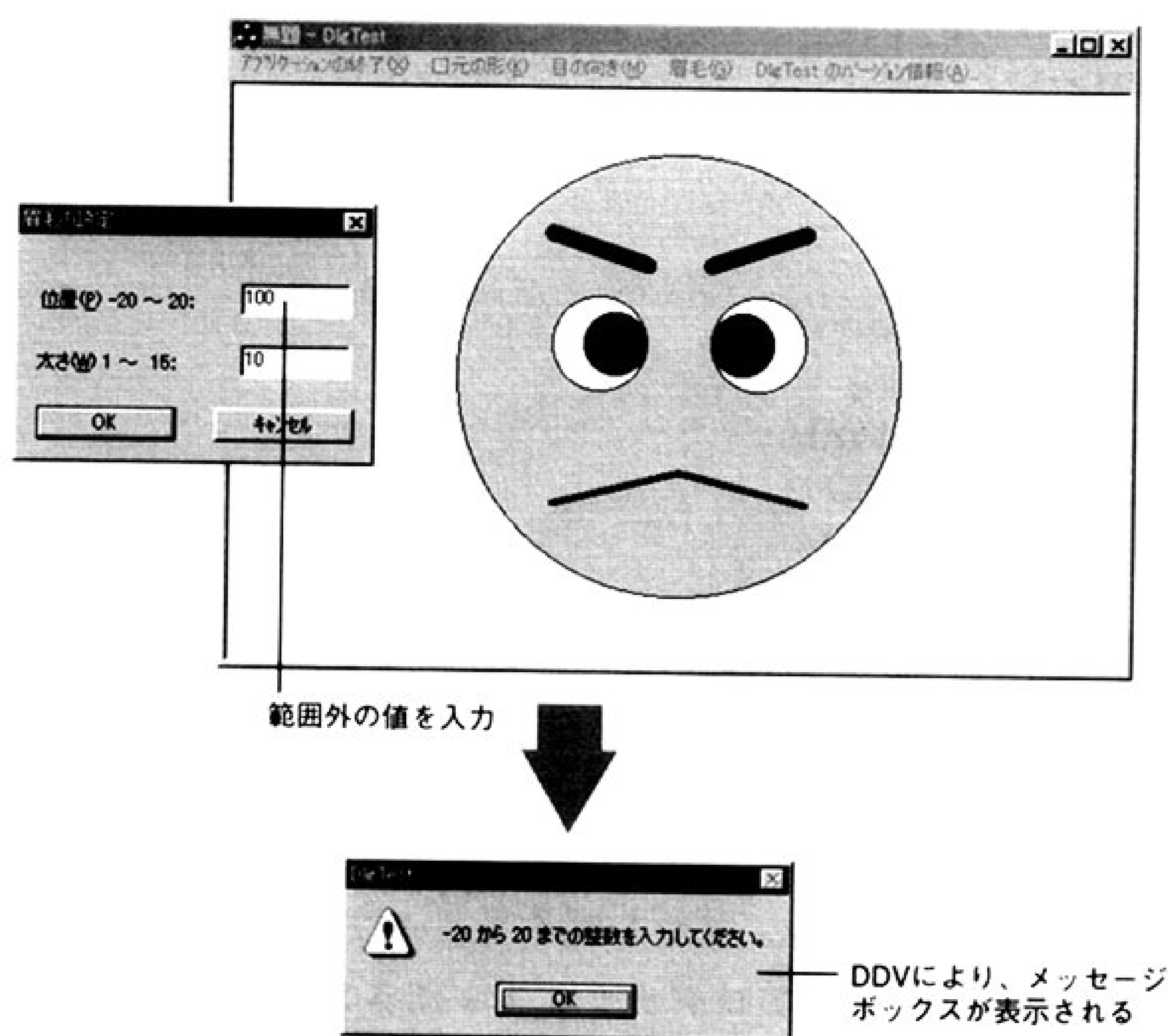


図 3-43　入力データが不正であったことを示すメッセージボックス

ここまで述べてきたように、エディットボックスでデータ入力を行うダイアログボックスを作成すると、ごく当然のように DDV の機能が組み込まれ、プログラム中でデータチェックを行う必要がほとんどなくなります。

4 マウスとキーボードからの入力

メニューやダイアログボックスを操作する場合には、「メニューが選択された」あるいは「コントロールの状態が変更された」というような、ある程度の意味を持ったメッセージが得られます。これはマウスボタンを押したりキーボードから文字を入力することで生じたメッセージをWindowsが処理してくれているおかげです。しかしこういった特別な処理の恩恵が受けられない部分、つまりクライアント領域の上では、プログラマが自力で生のマウスやキーボードからのメッセージを処理する必要があります。この章では、マウスメッセージを処理してマウス入力を行う方法と、キーボードメッセージを処理してキーボード入力を行う方法を説明します。

4.1 マウス入力

従来のMS-DOSプログラミングでは、マウスからの入力データは、プログラム側が「マウスの状態を調べるBIOS」を実行して読み取ることになっていました。

しかしWindowsでは、すべての入出力操作はメッセージを介して行われます。マウス入力に関しても例外ではありません。ユーザーがマウスを操作すると各種のマウスメッセージが発生し、アプリケーションはそのメッセージを受け取ってしかるべき処理を進めるのです。Windowsプログラミングとは、こちらからシステムを呼び出すのではなく、あくまでも送られてくるメッセージを待つものだという受け身の原則はここでも健在です。

本節ではマウスメッセージの扱い方を学ぶため、**図4-1**のようなToothPaste（練り歯磨き）グラフィックスを描くお絵かきプログラムを作ってみます。なお付録CD-ROMのPasteディレクトリには、このプログラムの完成版（次節で述べるキーボード入力機能を追加したもの）が収めてあります。

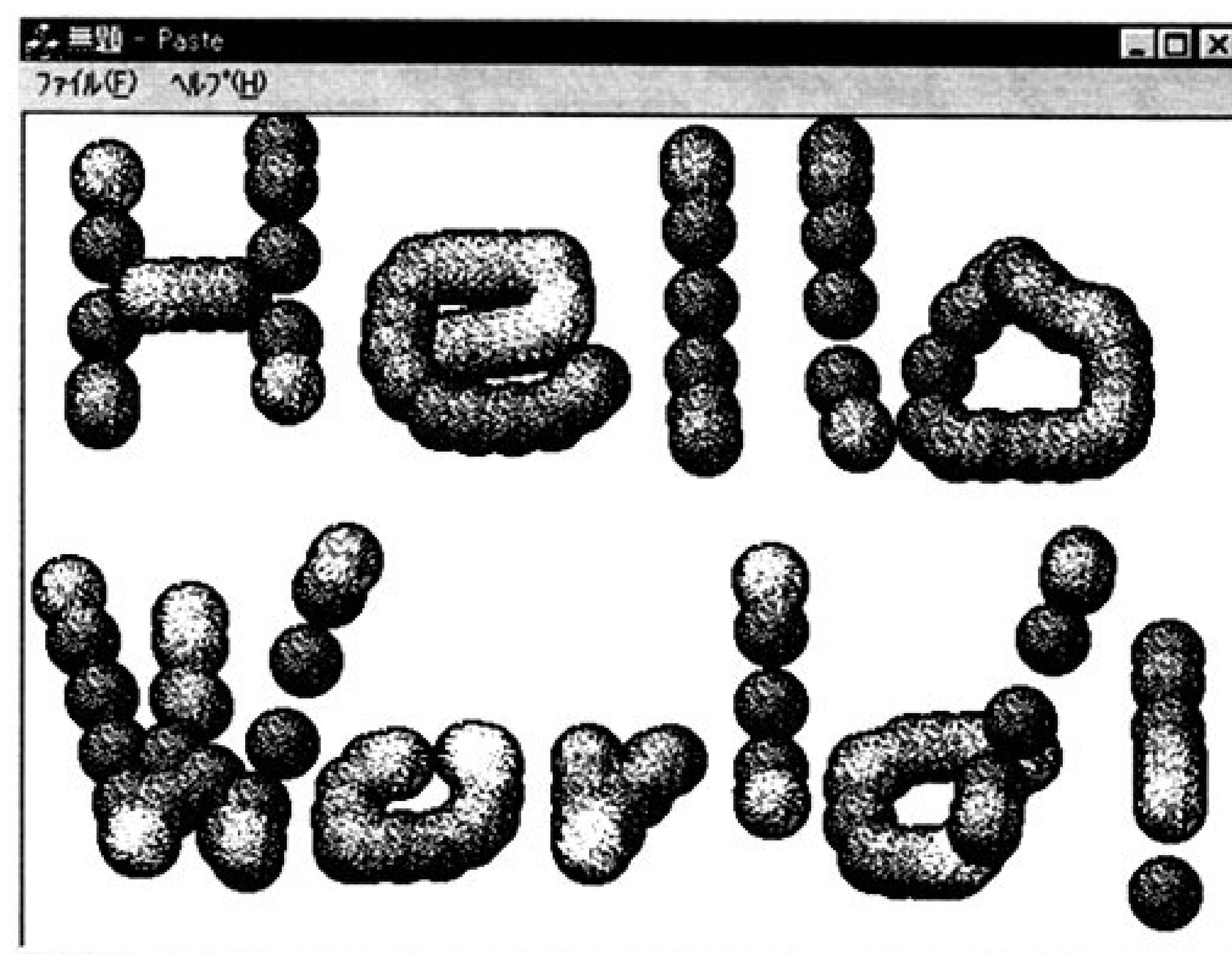


図 4-1 ToothPaste グラフィックス

スケルトンの作成

まずは AppWizard でプロジェクトの土台となるスケルトンを作成します。このプロジェクトには以下の設定を与えます。

- プロジェクト名：Paste
- アプリケーションタイプ：SDI
- データベースのサポート：しない
- 複合ドキュメントのサポート：しない
- ツールバー／ステータスバー：なし
- 印刷と印刷プレビュー：なし
- そのほか：デフォルトのまま

この条件に従って、AppWizard のオプションを設定してください。スケルトンが作られたら、リソースエディタを使用して、以下のようにメニューの不要項目を削除します。メニューの修正方法については、前出の MenuTest プロジェクトを参考にしてください。**図 4-2** にこのプロジェクトのメニュー画面を示します。

- ［ファイル］メニューは［アプリケーションの終了］だけ残してそれ以外を削除する
- ［編集］メニュー以下のメニューはすべて削除する



図 4-2 Paste プロジェクトのメニュー

● リソースの準備

次にToothPasteグラフィックスを描くための若干の準備を行います。ToothPasteグラフィックスは、画面だけ見るとウネウネのグニャグニャで、どのように描かれているのかわかりにくいのですが、描画の原理は非常に簡単です。

用意するものは**図4-3**に示すような黒い丸と白いモヤモヤの2つの図形です。黒丸をマウスの移動につれて即座に表示し、そのあとで少し遅れて白モヤを重ねていくと、あのウネウネが現れるという仕掛けです。

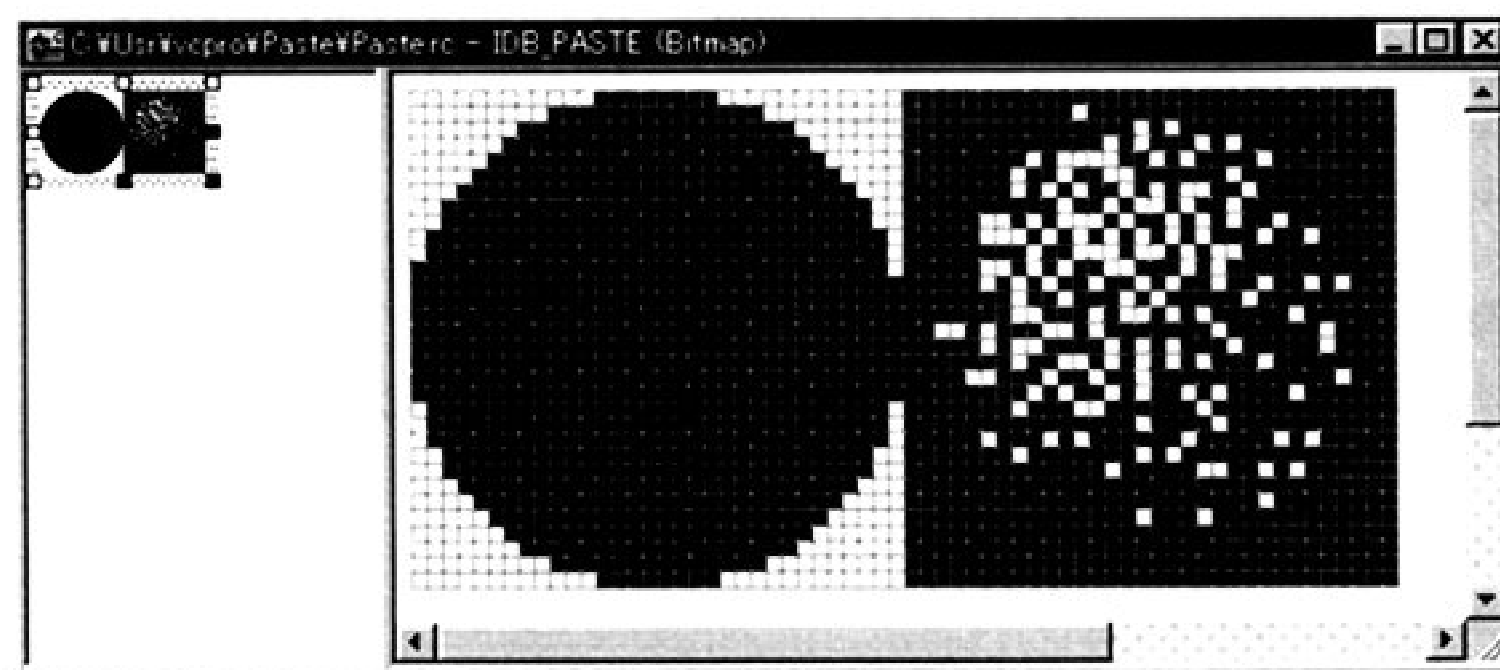


図4-3　ToothPasteグラフィックスのもとになる2つのビットマップ

2つの図形データはビットマップリソースとして用意します。ビットマップリソースの作成には、リソースエディタに含まれているグラフィックエディタを利用します。

まずは、メニューから[挿入]－[リソース]を選択するか、ワークスペースウィンドウの[ResourceView]ページで[Paste]フォルダを右クリックしてショートカットメニューから[挿入]を選択し、[リソースの挿入]ダイアログボックスを表示します。次に、[リソースの挿入]ダイアログボックスから[Bitmap]を選択し、<新規作成>ボタンをクリックします。するとグラフィックエディタが起動して、白い正方形のビットマップが表示されます。この正方形の枠線の右下をマウスでドラッグして横64ドット×縦32ドットのサイズに変更し、さらに楕円塗りつぶしツールなどを利用して上記の**図4-3**に示すような図形（左半分は白地に黒丸、右半分は黒地に白い点）を描いてください。

最後にビットマップの外側の部分をマウスでダブルクリックすると、ビットマップのプロパティが表示されます。ここでは[ID]ボックスに[IDB_PASTE]と入力します。これでリソースの準備は完了です。

● ウネウネを描く関数

次に黒丸と白モヤを時間差表示するためのサポート用の関数を2つ、PasteView.cppファイルの最後に追加します（**リスト4-1**）。これらの関数は、あとで作成するマウスメッセージハンドラの中から呼び出されます。

リスト 4-1　GrowLine 関数と ShrinkLine 関数（PasteView.cpp）

```
CObList PointList;

void GrowLine(CDC* pDC, CPoint point)
{
  CBitmap Bitmap, *pOldBitmap;
  CDC    MemDC;

  // point で示される座標に黒丸のビットマップを表示する
  Bitmap.LoadBitmap(IDB_PASTE);
  MemDC.CreateCompatibleDC(pDC);
  pOldBitmap = MemDC.SelectObject(&Bitmap);
  pDC->BitBlt(point.x - 16, point.y - 16, 32, 32, &MemDC,  0, 0, SRCAND);
  MemDC.SelectObject(pOldBitmap);

  // PointList の先頭に point を追加する
  PointList.AddHead((CObject*)(new CPoint(point)));
}

void ShrinkLine(CDC* pDC)
{
  CPoint* pTail;
  CBitmap Bitmap, *pOldBitmap;
  CDC    MemDC;

  // PointList の末尾から座標を取り出して白モヤを表示する
  pTail = (CPoint*) PointList.RemoveTail();
  Bitmap.LoadBitmap(IDB_PASTE);
  MemDC.CreateCompatibleDC(pDC);
  pOldBitmap = MemDC.SelectObject(&Bitmap);
  pDC->BitBlt(pTail->x - 16, pTail->y - 16, 32, 32, &MemDC, 32, 0, SRCPAINT);
  MemDC.SelectObject(pOldBitmap);
  delete pTail;
}
```

この２つの関数は、本節の目的であるマウス処理とはあまり関係がありません。ここではざっと動作を説明するだけにとどめておきましょう。

最初の GrowLine 関数は、引数 point で指定した座標に黒丸のビットマップを表示し、さらにその座標をあとでもう一度参照できるように、PointList というデータバッファの先頭に追加します。なおこの関数ではビットマップ表示を行う際に CDC::BitBlt 関数の第８引数として SRCAND オペレーションを指定しているため、ビットマップの白色の部分は背景がそのまま変更されずに残り、黒色部分だけが画面に書き込まれます。

２つ目の ShrinkLine 関数は、保存しておいた座標を PointList から取り出して、そこに白モヤのビットマップを表示します。こちらは CDC::BitBlt 関数で SRCPAINT オペレーションを指定しているため、ビットマップの黒色部分は背景のまま変わらず、白色部分だ

けが画面に表示されます。したがって、たとえばGrowLine関数を10回呼び出して黒丸を表示しておき、続けてShrinkLine関数を10回呼び出せば、10個の黒丸の上に順次白モヤが表示されていくわけです。

リスト4-1ではPointListにキュー*1の役割を持たせていますが、これを実現するために、バッファの先頭と末尾から自由にデータの追加や削除ができるCObListクラスを利用しました。CObListクラスの機能や使い方に関しては、オンラインマニュアルの「MFCリファレンス」を参照してください。

またGrowLine関数中の「new CPoint(point)」は、pointと等しい内容を持つCPointクラスの新しいオブジェクトを作成します。作成されたオブジェクトはあとでShrinkLine関数中の「delete pTail」によって削除されます。new演算子とdelete演算子は、C言語のmalloc関数によるメモリ確保とfree関数によるメモリ解放の作業を自動化したような機能を持ちます。

● マウスメッセージ

まずマウスから送られるメッセージの種類と、その使い方を簡単に説明します。

もっとも単純なマウスメッセージは、クライアント領域でボタンが押された瞬間に生じる**ボタンダウンメッセージ**と、離された瞬間に生じる**ボタンアップメッセージ**です。これらのメッセージはマウスの左右のボタンごとに用意され、それぞれ次のような名前が与えられています。

- **WM_LBUTTONDOWN**：左ボタンが押された
- **WM_LBUTTONUP**：左ボタンが離された
- **WM_RBUTTONDOWN**：右ボタンが押された
- **WM_RBUTTONUP**：右ボタンが離された

またクライアント領域でマウスカーソルが移動すると、次に示す**マウス移動メッセージ**が発生します。システムの処理速度によっても異なりますが、これはマウスの軌跡をあまり細かく表すものではありません。マウスを素早く移動させた場合など、WM_MOUSEMOVEメッセージの生じる間隔は、かなり飛び飛びになります。

- **WM_MOUSEMOVE**：マウスが移動した

マウスボタンをクリックすることは、Windowsのユーザーインターフェイスの基本ですが、クリック(押して離す)という動作そのものを表すマウスメッセージはありません。これはボタンダウンとボタンアップの組み合わせで簡単に判定できるからです。しかし**ダブ**

*1 キュー(Queue：先入れ先出しバッファ)。たとえばA、B、Cというデータをこの順番で書き込むと、同じくA、B、Cという順番で取り出すことができる。本プログラムのように、一連のデータを時間差を置いて読み書きしたい場合によく利用される。

ルクリックには、その動作を表す専用のメッセージが存在します。左右の各マウスボタンに、以下の名前のメッセージが対応します。

- **WM_LBUTTONDBLCLK**：左ボタンがダブルクリックされた
- **WM_RBUTTONDBLCLK**：右ボタンがダブルクリックされた

ダブルクリックメッセージは、一定時間内に一定範囲の四角形の中で、マウスボタンが続けて2回クリックされた場合に生じます。最初のクリックの時点では、それがダブルクリックの1回目なのか単なるクリックなのか判断できないため、通常のボタンダウンとボタンアップメッセージが送り出されます。そしてダブルクリックの2回目のクリックの際に、ボタンダウンメッセージの代わりにダブルクリックメッセージが発生します。つまりダブルクリックの際には、次の4つのメッセージが順に送られているわけです（左ボタンの場合）。

1. ボタンを押す：WM_LBUTTONDOWN
2. ボタンを離す：WM_LBUTTONUP
3. ボタンを押す：WM_LBUTTONDBLCLK（これがダブルクリックを示す）
4. ボタンを離す：WM_LBUTTONUP

ボタンダウン、ボタンアップ、マウス移動、ダブルクリックの各メッセージが送られる際には、かならず2種類の情報が付加されます。1つはマウスカーソルの位置、そしてもう1つは押されているマウスボタンとシフトキーの状態を示す情報です。ClassWizardで作成したマウスメッセージのメッセージハンドラには、この2つのデータが次のような引数として渡されます。

```
void CPasteView::OnLButtonDown(UINT nFlags, CPoint point)
{
  // WM_LBUTTONDOWN の処理
}
```

第1引数のnFlagsは、**表4-1**に示すビットフラグの論理和となっています。MK_MBUTTONフラグは3ボタンマウスのために用意されたもので、通常の2ボタンマウスでは常に0となります。またMK_SHIFTフラグやMK_CONTROLフラグはキーボードの状態を示しますが、これがマウスメッセージに含まれているのは、Windowsアプリケーションでは Shift や Ctrl を押しながらマウス操作を行うことが多いという、ごく実際的な理由からです。プログラミングの手間を省く（いちいちキーボードの状態を調べなくともよい）ための、ちょっとした心配りではあります。

フラグ	意味
MK_LBUTTON	左ボタンが押されている
MK_RBUTTON	右ボタン 〃
MK_MBUTTON	中ボタン 〃
MK_SHIFT	Shift 〃
MK_CONTROL	Ctrl 〃

表 4-1 nFlags の意味

マウスメッセージの第2引数 point は、メッセージが発生した瞬間のマウスカーソルの位置を示す CPoint クラスのデータです。クライアント領域の左上隅が原点 (0, 0) となります。CPoint クラスのデータは x 座標と y 座標を示す 2 つのメンバ変数で構成され、これ自体を 1 つの引数として扱うこともできますし、次のように 2 つのメンバ変数、x と y を利用して、x 座標や y 座標を個々に扱うことも可能です。

```
void CPasteView::OnLButtonDown(UINT nFlags, CPoint point)
{
  int x, y;

  x = point.x;   //point から x 座標を得る
  y = point.y;   //point から y 座標を得る
  ...
}
```

さて、マウスメッセージを処理して ToothPaste グラフィックスを描く手順は、要するに左ボタンが押されている間はマウスカーソルの移動に合わせて図形を描き、左ボタンが離されたら描画をやめる、簡単にいえばこれだけのことです。そこで ClassWizard を使って、WM_LBUTTONDOWN メッセージ、WM_MOUSEMOVE メッセージ、WM_LBUTTONUP メッセージに、それぞれ**リスト 4-2** のようなメッセージハンドラを割り当てます。なお処理内容が画面表示にかかわるものなので、ハンドラは CPasteView クラスに作成します。クラス名/オブジェクト ID/メッセージの組は**表 4-2** に示します。

クラス名	オブジェクト ID	メッセージ	メッセージハンドラ
CPasteView	なし(CPasteView を選択)	WM_LBUTTONDOWN	OnLButtonDown
CPasteView	なし(CPasteView を選択)	WM_MOUSEMOVE	OnMouseMove
CPasteView	なし(CPasteView を選択)	WM_LBUTTONUP	OnLButtonUp

表 4-2 ToothPaste グラフィックを描くメッセージハンドラ

リスト4-2 マウスメッセージのメッセージハンドラ(PasteView.cpp)

```
BOOL ButtonDown = FALSE;

void CPasteView::OnLButtonDown(UINT nFlags, CPoint point)
{
  CDC* pDC = GetDC();
  GrowLine(pDC, point);           // マウスカーソルの位置に黒丸を表示する
  ReleaseDC(pDC);
  ButtonDown = TRUE;              // 左ボタンが押されていることを示す
}

void CPasteView::OnMouseMove(UINT nFlags, CPoint point)
{
  CDC* pDC = GetDC();
  if (ButtonDown)                 // 左ボタンが押されていれば
    GrowLine(pDC, point);         //   マウスカーソルの位置に黒丸を表示する
  if (PointList.GetCount() >= 10) // キューの長さが10以上なら
    ShrinkLine(pDC);              //   白モヤに黒丸の後を追いかけさせる
  ReleaseDC(pDC);
}

void CPasteView::OnLButtonUp(UINT nFlags, CPoint point)
{
  CDC* pDC = GetDC();
  while (!PointList.IsEmpty())    // キューに残っている座標の後始末
    ShrinkLine(pDC);
  ReleaseDC(pDC);
  ButtonDown = FALSE;             // もう左ボタンは押されていないことを示す
}
```

このプログラムのポイントは、左ボタンの状態を示すためにButtonDownというフラグを用意して、WM_LBUTTONDOWNメッセージハンドラと、WM_LBUTTONUPメッセージハンドラによって、このフラグの値をTRUEあるいはFALSEに設定しているところです。これはマウス入力を処理する常套手段でもあり、覚えておくとよいでしょう。

次にダブルクリックメッセージを処理する例として、右ボタンのダブルクリックで画面がクリアされる機能をこのプログラムに追加してみましょう。こちらの動作は非常に単純で、WM_RBUTTONDBLCLKメッセージを処理する**リスト4-3**のようなハンドラをCPasteViewクラスに追加するだけです。メッセージハンドラは**表4-3**のような指定で作成してください。

クラス名	CPasteView
オブジェクト ID	なし(CPasteView を選択)
メッセージ	WM_RBUTTONDBLCLK

表 4-3 右ボタンのダブルクリックを処理するメッセージハンドラ

リスト 4-3 右ボタンのダブルクリックを処理するメッセージハンドラ(PasteView.cpp)

```
void CPasteView::OnRButtonDblClk(UINT nFlags, CPoint point)
{
  CDC*  pDC = GetDC();
  CRect rect;
  CBrush  eraser(GetSysColor(COLOR_WINDOW));  //画面消去用ブラシの用意

  GetClientRect(&rect);              //クライアント領域の座標を得る
  pDC->FillRect(&rect, &eraser);  //eraser ブラシで塗りつぶす
  ReleaseDC(pDC);
}
```

リスト 4-3 は、CDC::FillRect 関数を利用して、クライアント領域全体を eraser に指定したブラシで塗りつぶします。eraser ブラシはシステムカラー COLOR_WINDOW(ウィンドウ背景色)のソリッドブラシです。このようにシステムカラーを利用すると、Windows のコントロールパネルでウィンドウ背景色がどのように設定されていても、かならずその色で画面クリアが行われることになります。

● マウスのキャプチャー

マウスメッセージは、通常はマウスカーソルの真下のウィンドウに向けて送られます。いい換えると、ウィンドウがマウスメッセージを受け取れるのは、そのウィンドウ上にマウスカーソルがある場合に限られます。

たとえば前項の Paste プログラムを使って、こんな実験をしてみましょう。まずボタンを押したままマウスカーソルをクライアント領域の外まで動かします。そしてそこでボタンを離し、再びクライアント領域の中にマウスカーソルを戻します。するとボタンを押していないにもかかわらず、マウスの移動につれて図形が描かれてしまいます(**図 4-4**)。

これは、前項の Paste プログラムが、「押したボタンはそのウィンドウの上でかならず離される」ことを前提として ButtonDown の値を管理していたからです。ところがクライアント領域の外部でボタンを離すと、ビューオブジェクトにボタンアップ(WM_LBUTTONUP)メッセージが送られないため、CPasteView::OnLButtonUp 関数が実行されず、ButtonDown 変数の値が正しく更新されません。

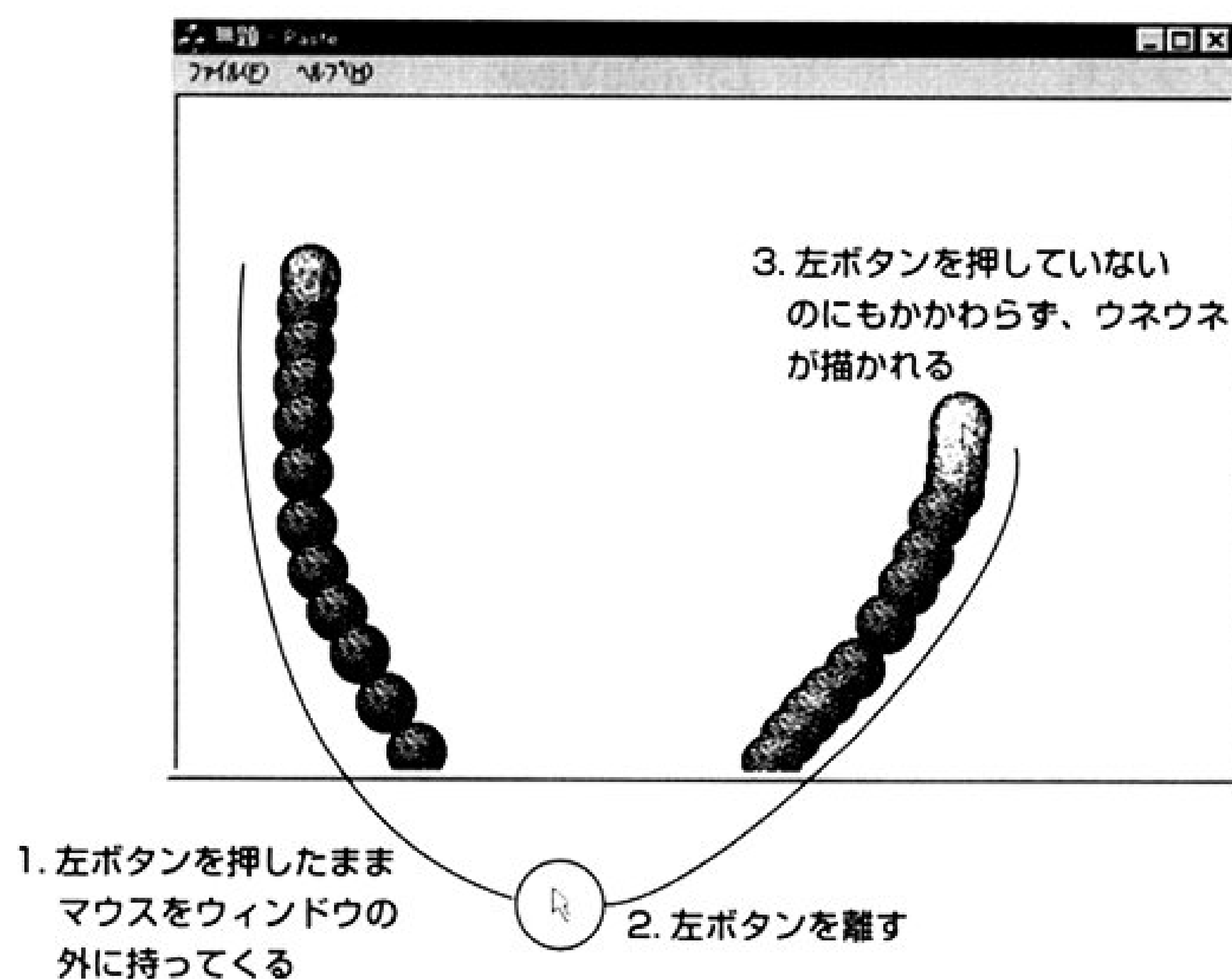


図 4-4　クライアント領域外でボタンを離すと不整合が起きる

このような問題を避けるために、**マウスのキャプチャー**（Capture：捕獲）という機構が用意されています。これは簡単にいうと、特定のウィンドウにマウスを占有させ、すべてのマウスメッセージを受け取らせようというものです。

マウスをキャプチャーしたウィンドウは、ディスプレイ上のどこにマウスカーソルが位置していても、かならずマウスメッセージを受け取ることができます。その代わり他のウィンドウにはマウスメッセージがいっさい送られなくなります。

この Paste プログラムの場合なら、左ボタンが押された（WM_LBUTTONDOWN を受けた）時点でビューオブジェクトがマウスをキャプチャーしてしまえば、あとはマウスがどこにあろうともかならず WM_LBUTTONUP メッセージを受け取ることができるので、確実に ButtonDown 変数の値が更新できます。ただしキャプチャーしたままでは画面上の他のアプリケーションの実行が不可能になりますから、ボタンが離されたらキャプチャーをリリース（解除）することも忘れてはいけません。

キャプチャーの開始と終了には、CWnd::SetCapture 関数および ReleaseCapture 関数を使います（ReleaseCapture 関数は MFC の関数ではなく Windows API）。マウスのキャプチャー／リリースを使ったプログラムは**リスト 4-4** のようになります。アミがかかった部分が追加した箇所です。

リスト4-4 マウスキャプチャーの修正を加えたメッセージハンドラ(PasteView.cpp)

```
void CPasteView::OnLButtonDown(UINT nFlags, CPoint point)
{
  CDC* pDC = GetDC();

  GrowLine(pDC, point);
  ReleaseDC(pDC);
  ButtonDown = TRUE;
  SetCapture();       //←キャプチャー開始
}

void CPasteView::OnLButtonUp(UINT nFlags, CPoint point)
{
  CDC* pDC = GetDC();

  while (!PointList.IsEmpty())
    ShrinkLine(pDC);
  ReleaseDC(pDC);
  ButtonDown = FALSE;
  ReleaseCapture();   //←キャプチャー終了
}
```

上記の変更を加えたプログラムをコンパイルし、実行してみてください。これならば、マウスをビューウィンドウの外に移動しても、さらにそこでボタンを離しても、プログラムの動作に支障はありません。

● マウスカーソルの位置の取得と変更

マウスカーソルの位置をプログラム中で読み取ったり変更することも可能です。現在のマウスカーソルの位置を調べるには、Windows APIとして提供されているGetCursorPos関数を利用します。この関数を次のように実行すると、POINT構造体の変数point(MFCを利用したプログラミング環境ではCPointクラスのオブジェクトを使ってもよい)に、マウスカーソルの位置が得られます。

```
POINT point;      //「CPoint point」としてもよい
GetCursorPos(&point);
```

また次のようにSetCursorPos関数(これもWindows API)を実行すると、指定した座標にマウスカーソルが移動します。この場合はPOINT構造体としてではなく、x座標とy座標を別々に与えます。

```
int x, y;
SetCursorPos(x, y);
```

GetCursorPos関数やSetCursorPos関数で扱うマウスカーソルの位置は、スクリーン座標と呼ばれ、画面の左上隅を原点とします。このスクリーン座標と、実際のウィンドウのクライアント領域の座標（クライアント座標）を相互に変換するために、CWnd::ScreenToClient関数とCWnd::ClientToScreen関数という関数も用意されています。

CWnd::ScreenToClient関数は、スクリーン座標をクライアント座標に変換します。たとえばCMyViewというビュークラスがあるとき、次の例はマウスカーソルが指す位置に点を打ちます。

```
void CMyView::StoC()
{
  CDC*  pDC = GetDC();
  POINT point;

  GetCursorPos(&point);              // マウスカーソルのスクリーン座標を得る
  ScreenToClient(&point);            // クライアント座標に変換する
  pDC->SetPixel(point.x, point.y);  // 指定した座標に点を打つ
  Release(pDC);
}
```

その逆にCWnd::ClientToScreen関数はクライアント座標をスクリーン座標に変換します。次の例はクライアント領域の左端から10ピクセル、上端から20ピクセルの位置に、マウスカーソルを移動します。

```
void CMyView::CtoS()
{
  CDC*  pDC = GetDC();
  POINT point;

  point.x = 10;                      // クライアント座標として(10,20)を指定
  point.y = 20;
  ClientToScreen(&point);            // スクリーン座標に変換する
  SetCursorPos(point.x, point.y);   // マウスカーソルを指定座標に移動する
}
```

これらの関数は、キーボードでマウスの動作をシミュレートするような場合に、しばしば必要になります。具体例は次節で紹介します。

4.2 キーボード入力

MS-DOSのプログラミングでは、キーボード入力といえばgetchar関数などの1文字入力関数の出番でした。しかしWindowsプログラミングではキーボード入力を得るにも、やはりメッセージを処理しなければなりません。従来の「キーボード入力」という言葉から受けるイメージとは少々異なる発想が必要です。

キーボードの操作によって発生するキーメッセージは、大きく2種類に分けられます。1つはキーの上げ下げを知るための**キーストロークメッセージ**、もう1つは押されたキーの文字コードを知るための**文字コードメッセージ**です。

ここでは、前節のPASTEプログラムにキーボード入力機能を追加することで、キーボードメッセージの意味と利用方法を見てみることにします。

● キーストロークメッセージ

Visual C++では以下に示す2つのキーストロークメッセージが利用できます。なおWindowsには、このほかに、WM_SYSKEYDOWN、WM_SYSKEYUPというシステムキー（Alt を押しながら入力するキー）を処理するためのメッセージも存在しますが、ClassWizardはこれらのメッセージについてはサポートしていません。

- **WM_KEYDOWN**：キーが押された
- **WM_KEYUP**：キーが離された

キーストロークメッセージには、3つの情報が付加されます。たとえばWM_KEYDOWNを処理するメッセージハンドラを作成してみると、次のような引数が示されます。

```
void CPasteView::OnKeyDown(UINT nChar, UINT nRepCnt, UINT nFlags)
{
  // WM_KEYDOWN の処理
}
```

第1引数のnCharはキーを区別するキーコードです。キーボード上の個々のキーには、それぞれ異なるキーコードが与えられ、たとえばフルキーの 1 とテンキーの 1 はキーコードとしては別の値を持ちます。ただし、このキーコードはハードウェアの違いを吸収するために、**仮想キー**と呼ばれるWindowsの統一キー表現で示されます。

仮想キーの中には特定のハードウェアでのみ意味を持つものもありますが、**表4-4**に示す仮想キーは**標準キー**として、どんなWindowsシステムでも、対応する実際のキーがかならず存在することが保証されています。標準キー以外のキー（拡張キー）を含む仮想キー一覧はインクルードファイルWindows.hを参照してください。

仮想キー	キートップの対応
VK_CANCEL	Ctrl + Break
VK_BACK	Back Space
VK_TAB	Tab
VK_RETURN	Enter
VK_SHIFT	Shift
VK_CONTROL	Ctrl
VK_MENU	Alt
VK_CAPITAL	Caps Lock
VK_ESCAPE	Esc
VK_SPACE	（スペース）
VK_PRIOR	Page Up
VK_NEXT	Page Down
VK_END	End
VK_HOME	Home
VK_LEFT	←
VK_UP	↑
VK_RIGHT	→
VK_DOWN	↓
VK_INSERT	Insert
VK_DELETE	Delete
VK_0 ～ VK_9	0 ～ 9（フルキー）
VK_A ～ VK_Z	A ～ Z
VK_F1 ～ VK_F10	F1 ～ F10

表 4-4　仮想キーとキーボードのマッピング

メッセージハンドラの第2引数の nReptCnt は、同一のキーストロークが連続して発生した場合に、その回数を示します。たとえばキーボードのオートリピートによって特定のキー入力が連続して行われたとき、システムの処理が十分に速くキー入力ごとにメッセージの処理が可能ならば nReptCnt は1ですが、処理速度が遅い場合は nReptCnt に2以上の値が入ることがあります。

第3引数の nFlags は、**図 4-5** のようなビットフィールドから構成されます。この情報を利用することはほとんどありません。

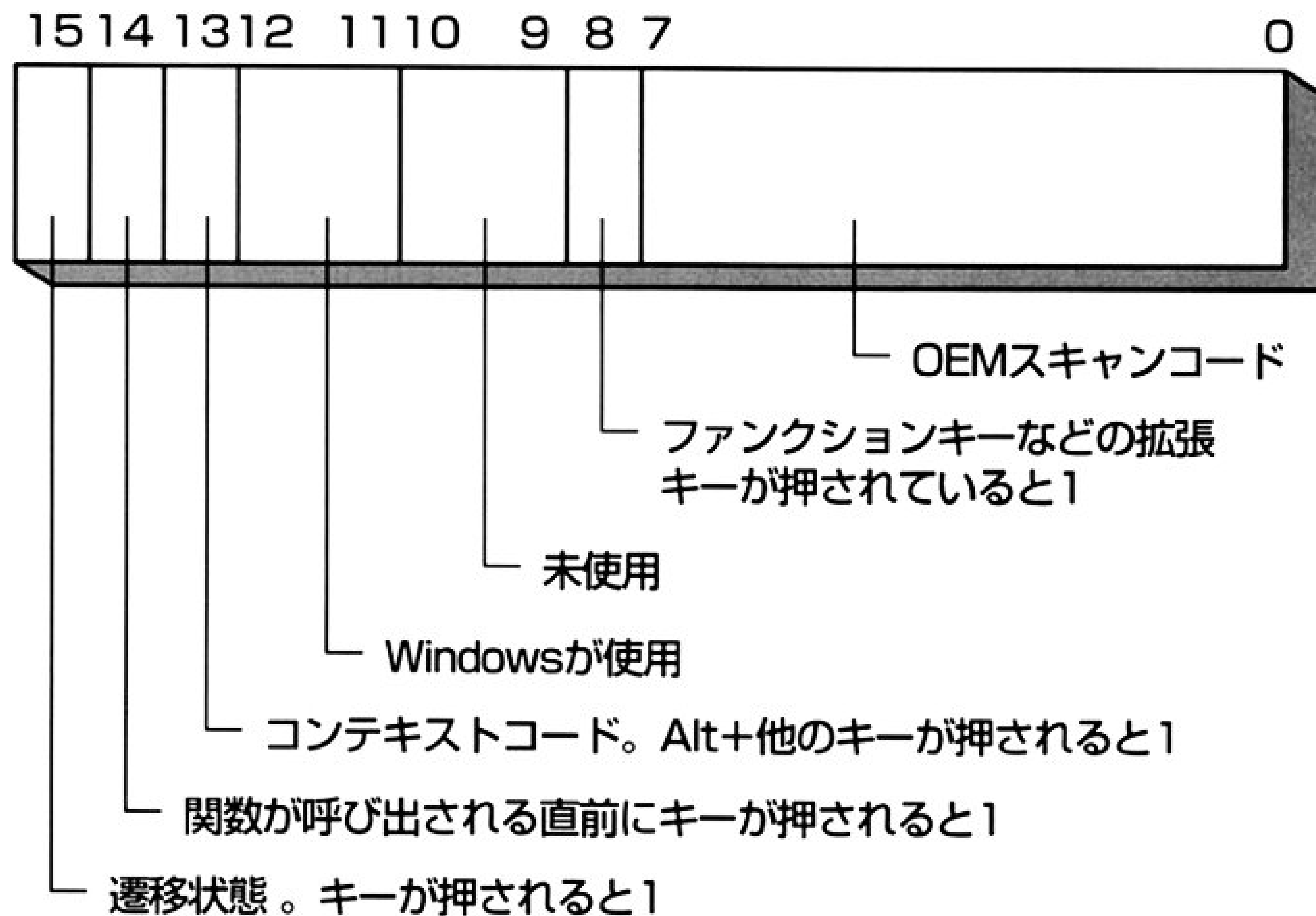


図 4-5 nFlags のビット構成

それでは、WM_KEYDOWN を利用して、PASTE プログラムにカーソルキーによるマウスカーソル移動の機能を追加してみましょう。**リスト 4-5** のようなメッセージハンドラを CPasteView クラスに対して作成してください。メッセージハンドラ作成時の指定は**表 4-5** に示します。メッセージハンドラの名前は「OnKeyDown」に一意に決まっているので省略します。

クラス名	CPasteView
オブジェクト ID	なし(CPasteView を選択)
メッセージ	WM_KEYDOWN

表 4-5 メッセージハンドラの指定

リスト 4-5 カーソルキーでマウスカーソルを移動させる(PasteView.cpp)

```
void CPasteView::OnKeyDown(UINT nChar, UINT nRepCnt, UINT nFlags)
{
  CDC*    pDC = GetDC();
  CPoint  point;

  GetCursorPos(&point);                // マウスカーソルの位置を得る
  switch (nChar) {
    case VK_LEFT:   point.x -= 3; break;   // [←] キー : 左移動
    case VK_RIGHT:  point.x += 3; break;   // [→] キー : 右移動
```

```
    case VK_UP:      point.y -= 3; break;  //[↑]キー：上移動
    case VK_DOWN:    point.y += 3; break;  //[↓]キー：下移動
    default:         break;                //その他のキー：何もしない
  }
  point.x += (rand() % 5 - 2);          //乱数を軽く加える
  point.y += (rand() % 5 - 2);
  SetCursorPos(point.x, point.y);       //新しい位置にマウスカーソルを移動する

  if (GetKeyState(VK_SHIFT) < 0) {      //[Shift] キーが押されていた場合
    ScreenToClient(&point);
    GrowLine(pDC, point);
    if (PointList.GetCount() >= 10) {
      ShrinkLine(pDC);
    }
  }
  else {                                //[Shift] キーが押されていない場合
    while (!PointList.IsEmpty())
      ShrinkLine(pDC);
  }
  ReleaseDC(pDC);
}
```

このメッセージハンドラをプログラムに組み込むと、カーソルキーでマウスカーソルを上下左右に動かせるようになります。またそのときに Shift を押していると画面に ToothPaste グラフィックスが表示されます。

Shift の判定には GetKeyState 関数（Windows API）を利用しています。この関数に VK_SHIFT という引数を与えると、現在の Shift の状態を調べ、押されていると戻り値は負の整数になります。このように、GetKeyState 関数はキーストロークメッセージとは独立に Shift の状態を読み取るのに便利な関数です。

● 文字コードによるキー入力

キーストロークメッセージは個々の物理的なキーを区別するものであり、文字入力に使うのにはやや不便です。たとえば仮想キー VK_1 は、Shift が押されていなければ数字の 1 を示しますが、Shift が同時に押されていれば ! を意味するわけです。

そこで、キーボードのシフト状態も考慮した実際の文字コードを得るために用意されたものが、次に示す WM_CHAR メッセージです。

- **WM_CHAR**：入力された文字の判定

WM_CHAR 用のメッセージハンドラは、キーストロークメッセージの場合と同様に、3つの引数を持ちます。

```
void CPasteView::OnChar(UINT nChar, UINT nRepCnt, UINT nFlags)
{
  // WM_CHAR の処理
}
```

この場合、第1引数nCharには、仮想キーコードではなく実際の文字コードが与えられます。第2引数のnRepCntと第3引数のnFlagsは、キーストロークメッセージの引数とまったく同じ意味を持ちます。

ここでは、[h]、[j]、[k]、[l]という4つのキー*2をWM_CHARメッセージハンドラで判定し、前項の矢印キーで行ったものと同様なカーソル移動を実現してみます（**リスト4-6**）。WM_CHARメッセージハンドラでは、[Shift]の状態が文字コードの違いとなって現れ、たとえば小文字の「h」と大文字の「H」のような区別ができるところに注目してください。メッセージハンドラを作成する際の指定は**表4-6**を参照してください。

クラス名	CPasteView
オブジェクトID	なし（CPasteViewを選択）
メッセージ	WM_CHAR

表4-6　WM_CHARメッセージを処理するメッセージハンドラ

リスト4-6　hjklでマウスカーソルを移動させる（PasteView.cpp）

```
void CPasteView::OnChar(UINT nChar, UINT nRepCnt, UINT nFlags)
{
  CDC*    pDC = GetDC();
  CPoint  point;
  BOOL    shift;

  GetCursorPos(&point);
  switch (nChar) {
    case 'h': point.x -= 3; shift = FALSE;  break;  // [h]：左
    case 'l': point.x += 3; shift = FALSE;  break;  // [l]：右
    case 'k': point.y -= 3; shift = FALSE;  break;  // [j]：下
    case 'j': point.y += 3; shift = FALSE;  break;  // [k]：上
    case 'H': point.x -= 3; shift = TRUE;   break;  // [H]：[Shift]+左
    case 'L': point.x += 3; shift = TRUE;   break;  // [L]：[Shift]+右
    case 'K': point.y -= 3; shift = TRUE;   break;  // [J]：[Shift]+下
    case 'J': point.y += 3; shift = TRUE;   break;  // [K]：[Shift]+上
    default:  break;
  }
  SetCursorPos(point.x, point.y);
```

*2 viキー、またの名をrogue（ローグ）キー。[h]、[j]、[k]、[l]がそれぞれ左、下、上、右へのカーソル移動を表す。

```
    if (shift) {  // [SHIFT] キーが押されていたらウネウネを描く
      point.x += (rand() % 5 - 2);
      point.y += (rand() % 5 - 2);
      ScreenToClient(&point);
      GrowLine(pDC, point);
      if (PointList.GetCount() >= 10) {
        ShrinkLine(pDC);
      }
    }
    else {           // [SHIFT] キーが押されていなければキューの処理を行う
      while (!PointList.IsEmpty())
        ShrinkLine(pDC);
    }
    ReleaseDC(pDC);
  }
```

5 デバッグしてみよう

本章ではこれまでとは少しばかり趣を変えて、デバッグに話題を移します。プログラマにとって、アプリケーションの設計を終えて、実際にコーディングをしているときがもっとも楽しく、クリエイティブな作業をしていることを実感できる瞬間でしょう。しかしアプリケーションは集中的なバグ出しとデバッグが行われるまでは完成品とは呼べません。デバッグ前の状態では、絵でいえばラフスケッチ、文章でいえば下書きが済んだ状態にすぎないのです。

Visual C++6.0 では、ツール（デバッガ）とライブラリ（MFC）の両面からバグを取り除き、バグの発生しにくいコードを記述するためのサポートがなされています。本章の目的はデバッグテクニックの紹介ではありませんから、一般的なデバッグ手法の解説はありません。Visual C++のデバッガの基本的な使い方と、MFC に用意されたデバッグ用クラスや関数の使い方を解説します。これまでソースコードとにらめっこばかりしてデバッグしていた方も、ちょっと遠回りにはなりますが本章の知識を身に付けて、効率的にデバッグを行えるようになりましょう。

5.1 デバッガ

どんなに優秀なプログラマでも最初からバグのないコードを書くことはできません。優れたコードを書くプログラマは、デバッグにも長けているものなのです。

デバッグの際に、もっとも基本的かつ強力なツールとして役立つのがデバッガです。デバッガを使えば実行中のアプリケーションを指定した箇所で停止し、その時点の変数、メモリ、コールスタック（関数の呼び出し経路）の内容を参照することができます。これを使えば、いちいちコードの中に変数の値を出力するデバッグ用コードを埋め込む必要もありません。また停止した箇所から、ソースコードを1行1行ステップ実行して、そのたびにどのような変化が起こったのかを確かめることができます。さらにデバッガを使ってアプ

リケーションを実行しているときには、エラーでアプリケーションが終了してしまっても、どの箇所で異常が発生したのかをソースコードウィンドウに示してくれます。

デバッグの下ごしらえ

さきほどデバッガをツールと呼びましたが、Visual C++ではデバッガが統合環境であるDeveloper Studioに含まれていて、単独のツールとして存在しているわけではありません。メニューの[ビルド]－[デバッグの開始]－[実行]をクリックすることで、デバッガを使って、現在の作業中プロジェクトのデバッグが開始されます。ただし、デバッガを使うためにはあらかじめプロジェクトをデバッグ可能な状態に設定しておく必要があります。すなわち、プロジェクト構成をデバッグ用に設定します。

MFC AppWizardを使ってプロジェクトを作成した場合、2つのプロジェクト構成が自動的に用意されます。たとえば「bug」という名前のプロジェクトを作成した場合、「bug - Win32 Release」と「bug - Win32 Debug」という2つのプロジェクト構成が作成されます。現在、どちらの構成で作業をしているかは、ビルドツールバー（**図5-1**）かメニューの[ビルド]－[構成]（**図5-2**）で確認することができます。

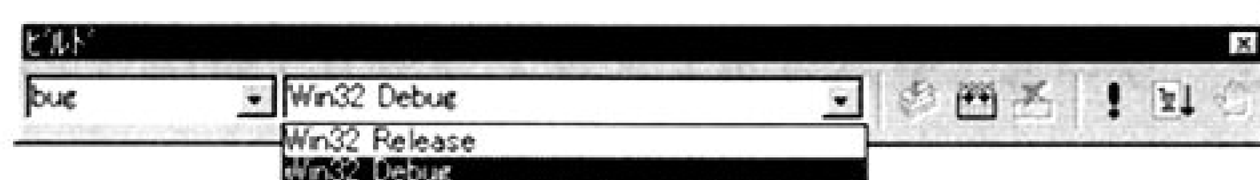


図5-1　ビルドツールバー

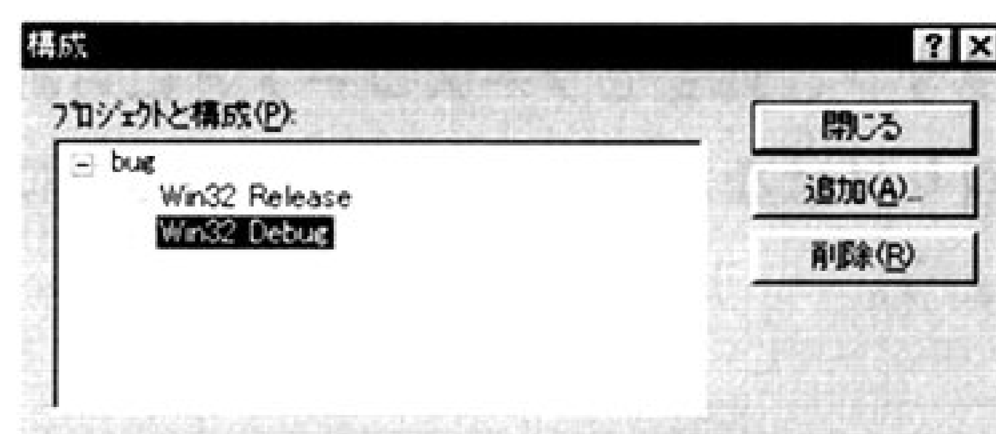


図5-2　プロジェクト構成

プロジェクトをビルドするときには、どちらかのプロジェクト構成を使うわけですが、デバッガを使ってデバッグするためには、[Win32 Debug]というプロジェクト構成でビルドをしておく必要があります。つまりデバッガを起動するためには、

1. ビルドツールバーで、または[ビルド]－[アクティブな構成を変更]を選択すると表示される[構成]ダイアログボックスでプロジェクト構成を「Win32 Debug」に変更
2. ビルドツールバーで、または[ビルド]－[ビルド]でプロジェクトをビルド
3. ビルドツールバーで、または[ビルド]－[デバッグの開始]－[実行]でデバッグを開始

という手順を踏むことになります。もっともMFC AppWizardでプロジェクトを作成した直後は［Win32 Debug］がアクティブに設定されているので、そのままデバッガを起動することができます。

Debug構成とRelease構成

ここで予備知識として、プロジェクト構成について解説しておきましょう。プロジェクト構成とはビルド時における設定をまとめたもので、これを切り替えることで必要に応じた実行ファイルを作成できるようにしています。たとえばMFC AppWizardを利用すれば、DebugとReleaseという2つのプロジェクト構成が作成されます。この2つのプロジェクト構成は、コーディング中はプロジェクト構成Debugを利用し、完成後にはプロジェクト構成Releaseを利用するというように使い分けると、目的に応じた実行ファイルが作成されるというわけです。

DebugとReleaseの2つのプロジェクト構成を使い分ける指針は極めて単純で迷うことのないものですが、なぜ使い分けが必要であるかは気になるところでしょう。プロジェクト構成の設定内容はメニューから［プロジェクト］－［設定］を選択すると表示される［プロジェクト設定］ダイアログボックスで確認することができるので、これを比較してみましょう。共通する設定項目については省略することにして、**表5-1**に異なる点についてのみまとめました。

設定項目	Debug	Release
出力ディレクトリ	Debug	Release
プリプロセッサの定義	_DEBUG	NDEBUG
デバッグ情報	生成する	生成しない
最適化	しない	する（実行速度優先）

表5-1　プロジェクト構成DebugとReleaseの違い

以上の違いについて、1つ1つ解説していくことにしましょう。

まず出力ディレクトリですが、これはビルド時に作られる.OBJファイルや.RESファイルといった中間ファイル、それに最終的に作成される実行ファイルの置き場所として使われるディレクトリです。これらのファイルはプロジェクト構成ごとに異なる内容で作られるため、別々のフォルダに保存されます。こうしておかないと、アクティブなプロジェクト構成を切り替えても、すべてのファイルのリコンパイルが行われずに、Debug用オブジェクトとRelease用オブジェクトが混在したままビルドされてしまうことがあるからです。

ソースファイル中では#defineディレクティブを使ってマクロを定義できますが、これと同じことをプロジェクト構成でも設定することができます。MFC AppWizardが生成したプロジェクト構成では、DebugとReleaseの両方であらかじめWIN32、_WINDOWS、

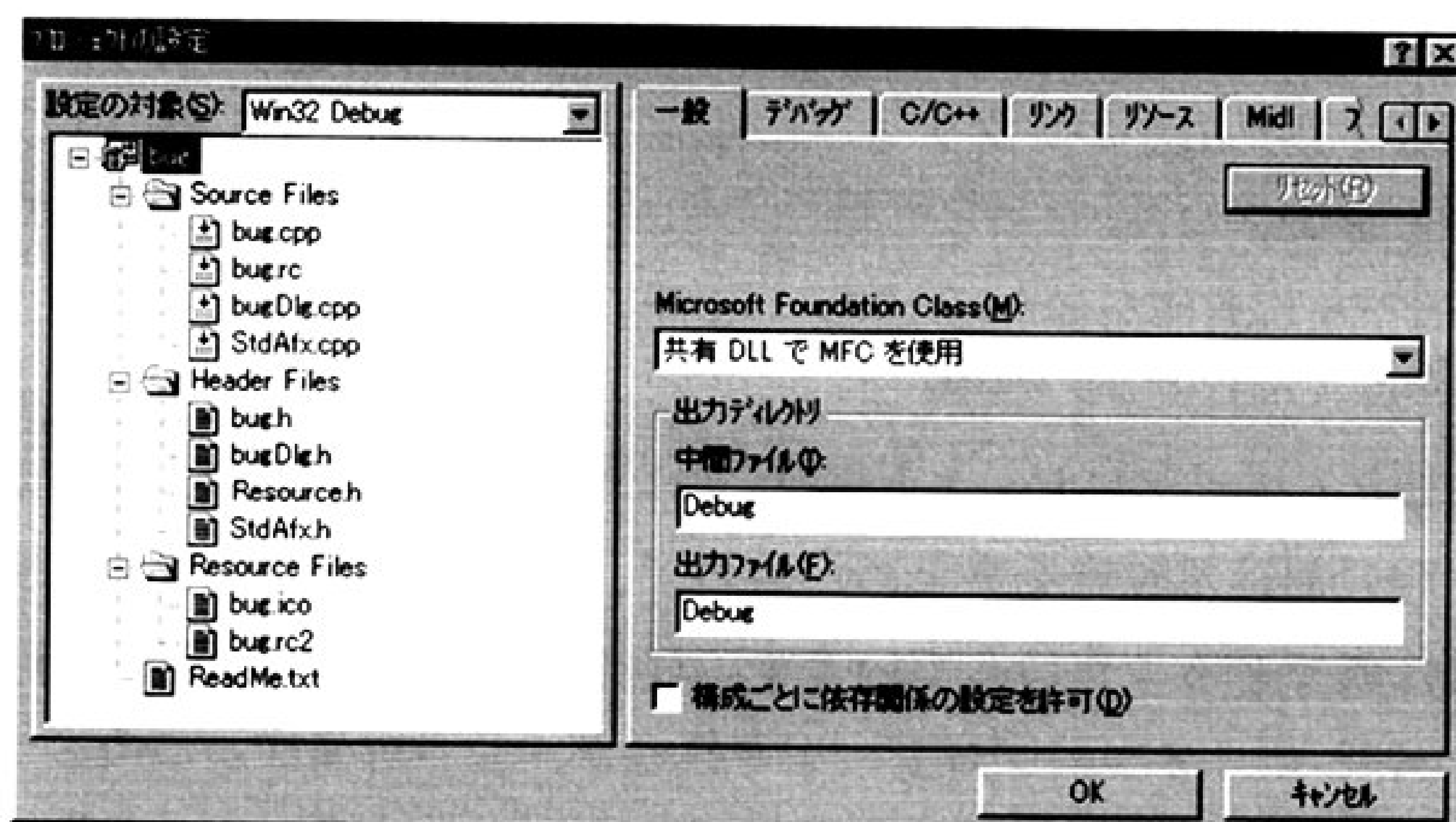


図 5-3 出力ディレクトリの設定（［一般］タブ）

_AFXDLL の 3 つのマクロがそれぞれ値を持たないマクロとして定義されています。これらは主にシステムインクルードファイルで参照されます。またこれに加えて Debug では_DEBUG が、Release では NDEBUG がそれぞれ定義されています。この 2 つのマクロを #ifdef ディレクティブで参照することによって、ソースコード中でどちらのプロジェクト構成でビルドされているのかを判断することができます。たとえば、MFC には CObject::Dump 仮想メンバ関数のようにプロジェクト構成 Debug でビルドするときにのみ定義されるメンバ関数が存在します。これらの仮想メンバ関数をオーバーライドするときにはかならず

```
#ifdef  _DEBUG
void CmyView::Dump(CDumpContext& dc)
{
...
}
#endif
```

のように_DEBUG マクロを参照します。とくに解説もせずに来ましたが、これまでに MFC AppWizard を使って作成したプロジェクトの中にはまさにこれと同じコードが含まれていたのです。

プロジェクト構成 Debug でビルドした実行ファイルは、デバッガを使ってソースファイルを参照しながらデバッグすることができます。これはビルド時に実行ファイルに加えて、デバッガが参照するデバッグ情報を生成しているためです。ただしデバッグ情報のサイズは大きくなりやすく、また実行ファイルのサイズも大きくなってしまうため、デバッグにしか利用しない情報を常に生成するのはディスクスペースの無駄となりますし、ビルドスピードも低下します。そこでプロジェクト構成 Debug のときにのみデバッグ情報を生成する設定がなされているのです。なおデバッグ情報は拡張子が PDB のファイルとして、出力ディレクトリに作成されます。

プロジェクト構成 Debug でのビルドは、プリプロセッサによるディレクティブ（#ifdef など）の処理、コンパイル（.OBJ ファイルの作成）、リンク（.EXE ファイルの作成）というステップで行われますが、Release によるビルドでは、リンクの手前に最適化と呼ばれる処理が行われます（最適化がサポートされるのは、Professional Edition 以上。Standard Edition ではサポートされない）。最適化の手法は非常に複雑であり、ここで簡単に説明できるものではありませんが、その目的はより実行速度がはやく、よりファイルサイズの小さな実行ファイルを作ることです。ただし最適化には時間がかかります。まだデバッグ中のコードであれば、実行速度の速い実行ファイルを作ることよりも、開発サイクルを短縮することの方が重要ですから、最適化は必要ありません。そのためプロジェクト構成 Release でのみ最適化が行われる設定がなされているのです。

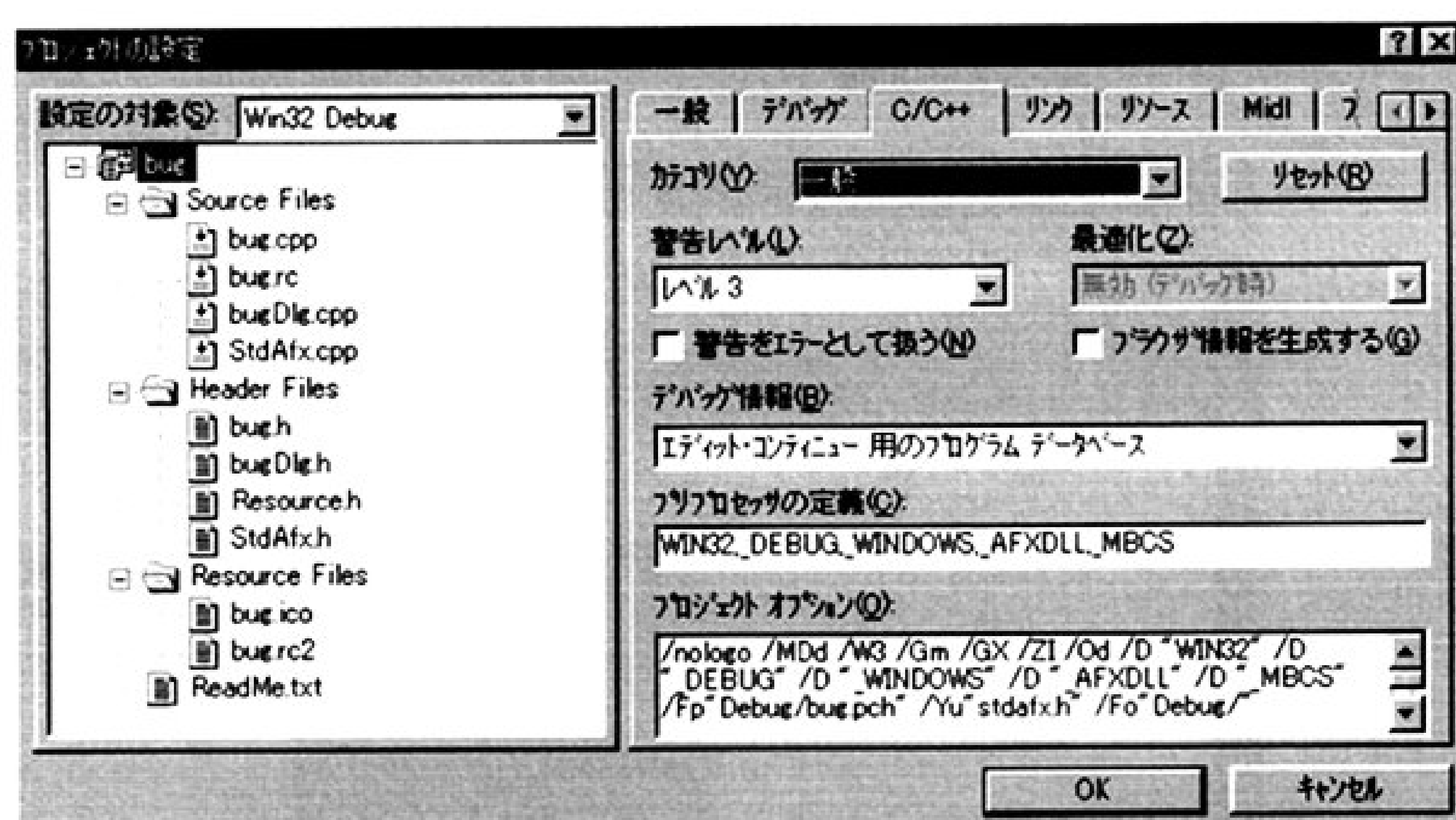


図 5-4 プリプロセッサの定義、デバッグ情報、最適化（[C/C++] − [一般]）

以上が 2 つのプロジェクト構成の違いです。プロジェクト構成の設定項目を変更しない限りは気にするべき項目はありませんが、マクロ_DEBUG と NDEBUG についてはあとで登場しますので覚えておいてください。

●ビルドエラー

それでは実際のデバッグ作業を通してデバッガの使い方を修得するために、1 つプロジェクトを作成しましょう。これまでと同じように MFC AppWizard を使って、次の手順に従って bug プロジェクトのスケルトンを作成してください。ここでは、ダイアログベースのアプリケーションを作成してみます。ダイアログベースのアプリケーションでは、ドキュメントービュー・アーキテクチャに依存しないプログラムを作成することができ、また Visual Basic のようにウィンドウ（フォーム）の上に直接コントロールを貼り付けていくことでユーザーインターフェイスを簡単に作成することができます。以降の処理は、ダイアログボックスの扱いとほぼ同様なので、ここでは詳しい説明は省きます。

1. 新規作成ダイアログボックスから［MFC AppWizard（exe）］を選択し、プロジェクト名にbugを入力する
2. MFC AppWizardのステップ1で、［作成するアプリケーションの種類］に［ダイアログベース］を選択
3. ＜終了＞ボタンをクリック

次にMFC AppWizardが生成したダイアログボックスIDD_BUG_DIALOGを編集します。ワークスペースウィンドウで［ResourceView］タブをクリックしてリソースの一覧を表示し、Dialogフォルダの下にあるIDD_BUG_DIALOGをダブルクリックしてください。ダイアログエディタが起動したら、**図5-5**および**表5-2**に示すように2つのエディットボックスと1つのボタンを配置します。

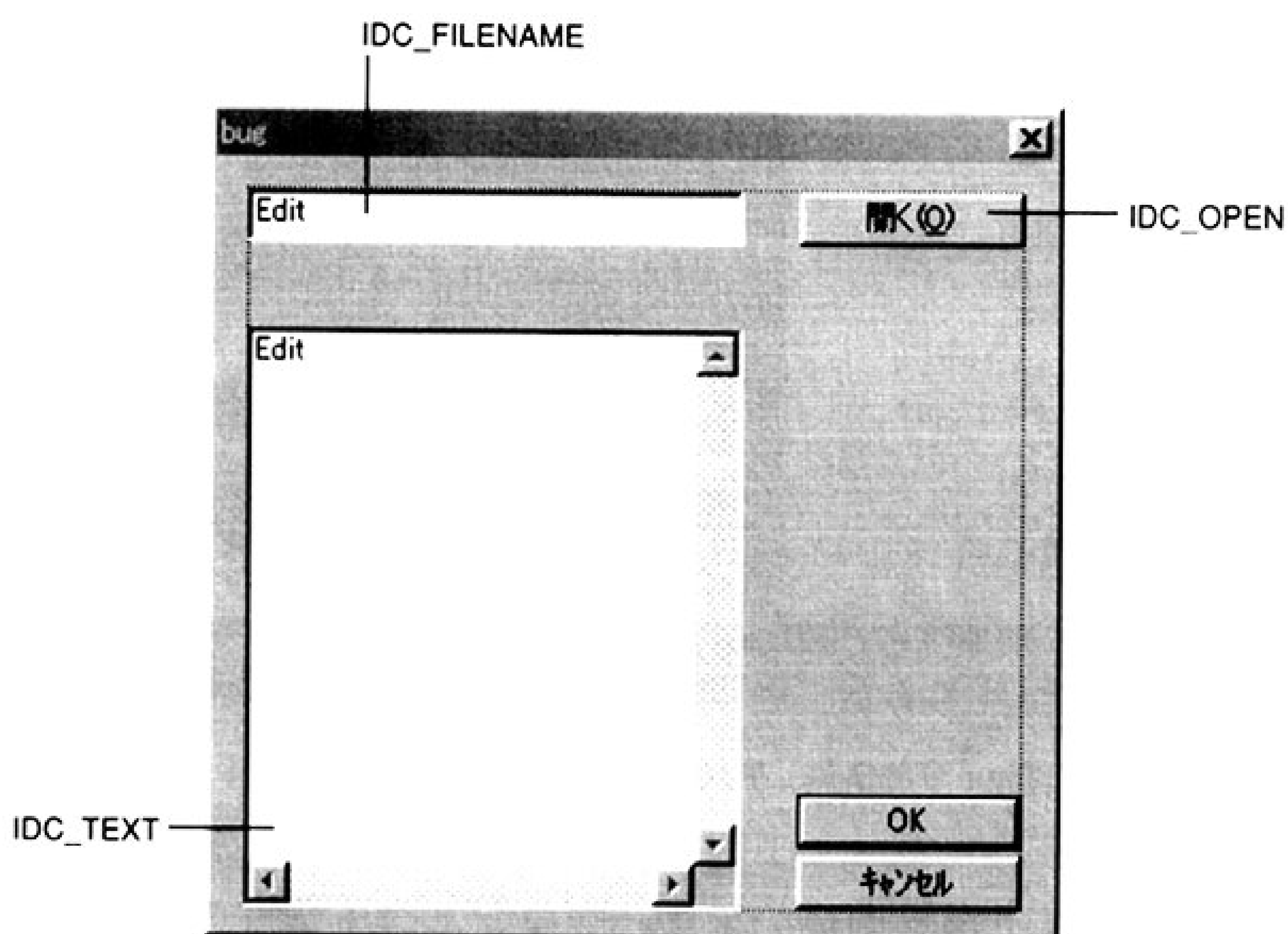


図5-5　IDD_BUG_DIALOG

アイテム	コントロール種	プロパティ
IDC_FILENAME	エディットボックス	変更なし
IDC_TEXT	エディットボックス	複数行、水平スクロールバー、水平オートスクロール、垂直スクロールバー、垂直オートスクロールの5つにチェック
IDC_OPEN	ボタン	キャプションに［開く（&O）］を入力

表5-2　コントロールの設定

配置したら、IDC_OPENボタンにメッセージハンドラを割り当てます。これにはWizardBarを使って、次の項目を選択したあとにコンテキストメニューから［Windowsメッセージハンドラの追加］を実行します。すると［クラスCBugDlg用のWindowsメッセー

ジおよびイベントハンドラの新規作成］ダイアログボックスが表示されるので、＜ハンドラの追加＞ボタンをクリックし、CBugDlg::OnOpen メッセージハンドラを作成します。

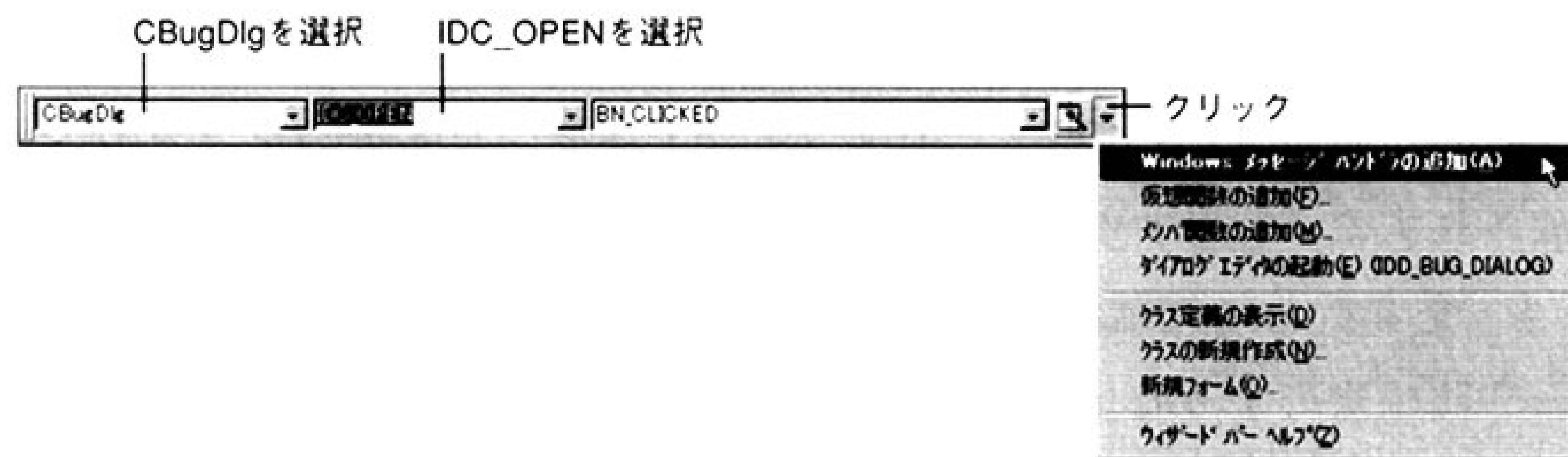


図5-6 WizardBar

次に作成された空のメッセージハンドラに**リスト 5-1** に示すコードを入力してください。ストーリーの都合上、意図的にバグを含ませてあります。気づいてしまった賢明な読者も多いとは思いますが、ここはひとまず黙っておいてください（簡単にいえば、**リスト 5-1** に示すとおりにコードを入力してください）。

リスト 5-1 CbugDlg::OnOpen メンバ関数（IDC_OPEN の BN_CLICKED メッセージハンドラ）

```
void CBugDlg::OnOpen()
{
  CFile f;
  CString filename;

  ((CEdit*)GetDlgItem(IDC_FILENAME)->GetWindowText(filename);
  if (!f.Open(filename, CFile::modeRead)) {
    AfxMessageBox(_T("ファイルが見つかりません"));
    return;
  }

  UINT length = f.GetLength();
  TCHAR* p = new TCHAR[length];

  if (f.Read(p, length) != length) {
    AfxMessageBox(_T("読み込みに失敗しました"));
    return;
  }

  ((CEdit*)GetDlgItem(IDC_TEXT))->SetWindowText(p);

  f.Close();
}
```

これでひとまずコーディングは終了です。このプログラムがうまく動けば、エディットボックス(IDC_FILENAME)にファイル名を入力し、<開く>ボタン(IDC_OPEN)をクリックするとOnOpen関数が呼び出されて、もう1つのエディットボックス(IDC_TEXT)に指定したファイルの内容が表示されるはずです。

早速[ビルド]-[ビルド]を実行してビルドしましょう。ビルド結果はアウトプットウィンドウに表示されます。もしアウトプットウィンドウが表示されていなければ、[表示]-[アウトプット]を実行して表示させてください。

残念ながら今回は次のようなメッセージが表示され、ビルドは失敗に終わっているはずです。

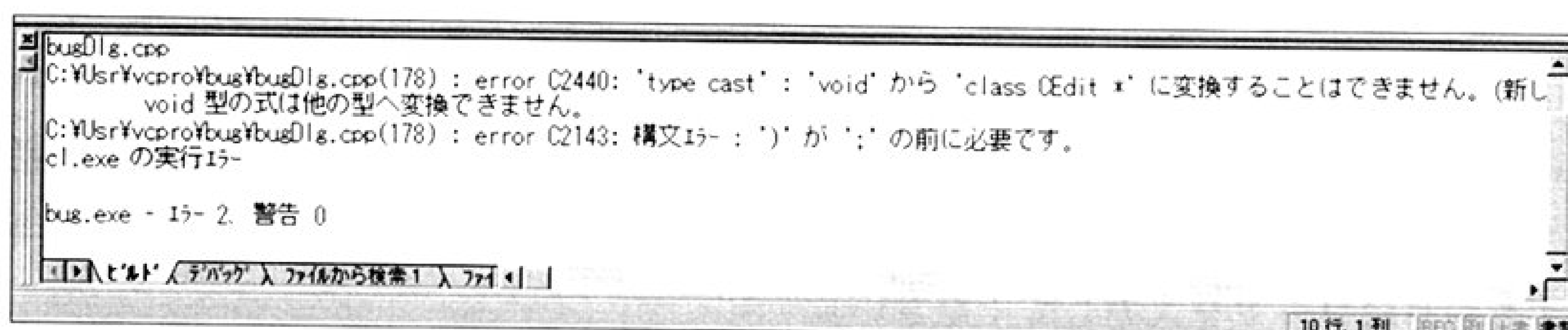


図5-7 ビルドエラー

これが本章最初の関門、「ビルドエラーメッセージの把握」です。ソースコードに誤りがあってビルドに失敗した場合、コンパイラがエラーメッセージを出力しますが、これがなかなかの曲者です。

まずエラーメッセージのフォーマットから解説することにしましょう。エラーメッセージはすべて次のフォーマットで出力されます。

<ソースファイル名>(<行番号>):<エラーコードまたは警告コード>:<コメント>

先のエラーメッセージの2行目ならば、ソースファイルC:¥Usr¥vcpro¥bug¥bugDlg.cppの178行目で、エラーC2143が発生したことを意味しています。そしてエラーC2143とは、構文エラーのことであり、今回は「(C++言語の文法に従うためには)パーレン(閉じカッコ)がセミコロンの前に必要(かもしれない)」と主張しているわけです。「セミコロンの前にパーレンを入力すれば解決する」と断言しているわけではないことに注意してください。

1行目のエラーメッセージも同じソースファイルの同じ行を指しているので、ともかくエラーが発生している行を表示させましょう。テキストエディタで行番号を指定してその周辺を表示させるためには[編集]-[ジャンプ]が使えますが、アウトプットウィンドウに表示された内容であれば、もっと簡単に F4 を入力するだけで、自動的にエラーメッセージが指す行にジャンプすることができます。あるいはエラーメッセージをダブルクリッ

クしてもかまいません。
　エラーが発生した行の内容は次のようなものです。

```
((CEdit*)GetDlgItem(IDC_FILENAME)->GetWindowText(filename);
```

　なぜ、この行でエラーが発生したのでしょうか? 先に正解を示してしまうと、この行は実は次のようになっていなければならないのです。つまり、パーレンをセミコロンの前ではなく、->の手前に入力する必要があります。

```
((CEdit*)GetDlgItem(IDC_FILENAME))->GetWindowText(filename);
```

　プログラムを書いているときには、GetDlgItem関数でエディットボックスを取得して、それに対してGetWindowText関数を実行し、ファイル名を取得すると考えていたわけです。なのに、なぜこのエラーが先のように報告されたのか、少々長くなりますがコンパイラの言い分を代弁してみましょう。
　次に示すのはコンパイラから見た178行目の内容です。コンパイラはまず1行(セミコロンまで)を分解可能なパーツに切り分けます。すると

```
( ( CEdit* ) GetDlgItem ( IDC_FILENAME ) -> GetWindowText ( filename )
```

のようになります。次にC++言語の文法に従ってパーツを組み合わせていきます。ここでは演算子の優先順位に従って、優先順位の高い順に組み合わせを進めていきます。まずここで登場するもっとも高い優先順位を持つ演算子は、関数呼び出しに使われるパーレンです(これも演算子の1つです)。そこでまずGetDlgItemメンバ関数とGetWindowTextメンバ関数の呼び出しに使われているパーレンをつなげます。結果は次のようになります。

```
( ( CEdit* ) GetDlgItem(IDC_FILENAME)  ->  GetWindowText(filename)
```

　これは問題ありません。次に高い演算子はメンバ参照に使われる->です。そこでGetDlgItemメンバ関数とGetWindowTextメンバ関数のブロックがつながります

```
(  (  CEdit*  )  GetDlgItem(IDC_FILENAME)->GetWindowText(filename)
```

　この時点ですでにプログラマの意図とは異なった解釈がなされていることがわかるでしょうか? 本来ならばGetDlgItemメンバ関数のCWnd*型の返り値をCEdit*型にキャストし、それからGetWindowTextメンバ関数を参照するはずでした。ところがキャストが行われる前に->による参照が行われてしまったため、CWnd::GetWindowTextメンバ関数が呼び出されると解釈されてしまっているのです。しかし誤った解釈がされているにもかかわらず、たまたまCWndクラスにもCEditクラスと同じGetWindowTextメンバ関数が存在したため、これをコンパイラはエラーと判断できなかったのです。
　そこでコンパイラは何食わぬ顔をして次の解釈に進みます。次は(CEdit*)によるキャスト

が処理されますが、この時点で初めてコンパイラはエラーを検出します。CWnd::GetWindowTextメンバ関数の返り値はvoidであるため、これをCEdit*型にはキャストできないからです(voidは返り値なしを意味しますから、存在しない値の型を変換することはできません)。これがエラーメッセージの1行目の正体です。さらに残った先頭のパーレンを解釈しようと試みますが、対応するパーレンは存在しないため、「最後にパーレンがないよ」とエラーメッセージの2行目が出力されたのです。

このようにコンパイラはできる限りC++言語の文法に忠実に解釈しようと試みます。これに失敗するとコンパイラはエラーメッセージを出力するのですが、すでに見たようにこれがかならずしもプログラマにとって直接有益な情報とは限らないのです。エラーメッセージを正確に読み取るためにはコンパイラの気持ちになって考えることです。このとき参考になるのが、エラーコードのヘルプドキュメントです。アウトプットウィンドウのエラーメッセージの行にカーソルを置いて F1 を入力すると、そのエラーが意味するところを解説したドキュメントが表示されます。例も交えて詳しく解説されているので、ぜひ活用してください。

完成？ちょっとまった！

バグの原因も解決方法も明らかになったところで、前節のバグを修正して、改めてビルドしてください。今度は無事ビルドも終了したことでしょう。そこでできあがった実行ファイルを、デバッガを使って起動します。このためには、メニューから[ビルド]－[デバッグの開始]－[実行]を選択します。すると普通に起動した場合と同じように、プロジェクトbugのダイアログボックスが表示されます。ここで**図5-8**に示すような操作をしてください。

すると指定したファイルの内容が、下のエディットボックスに表示されたはずです。これ

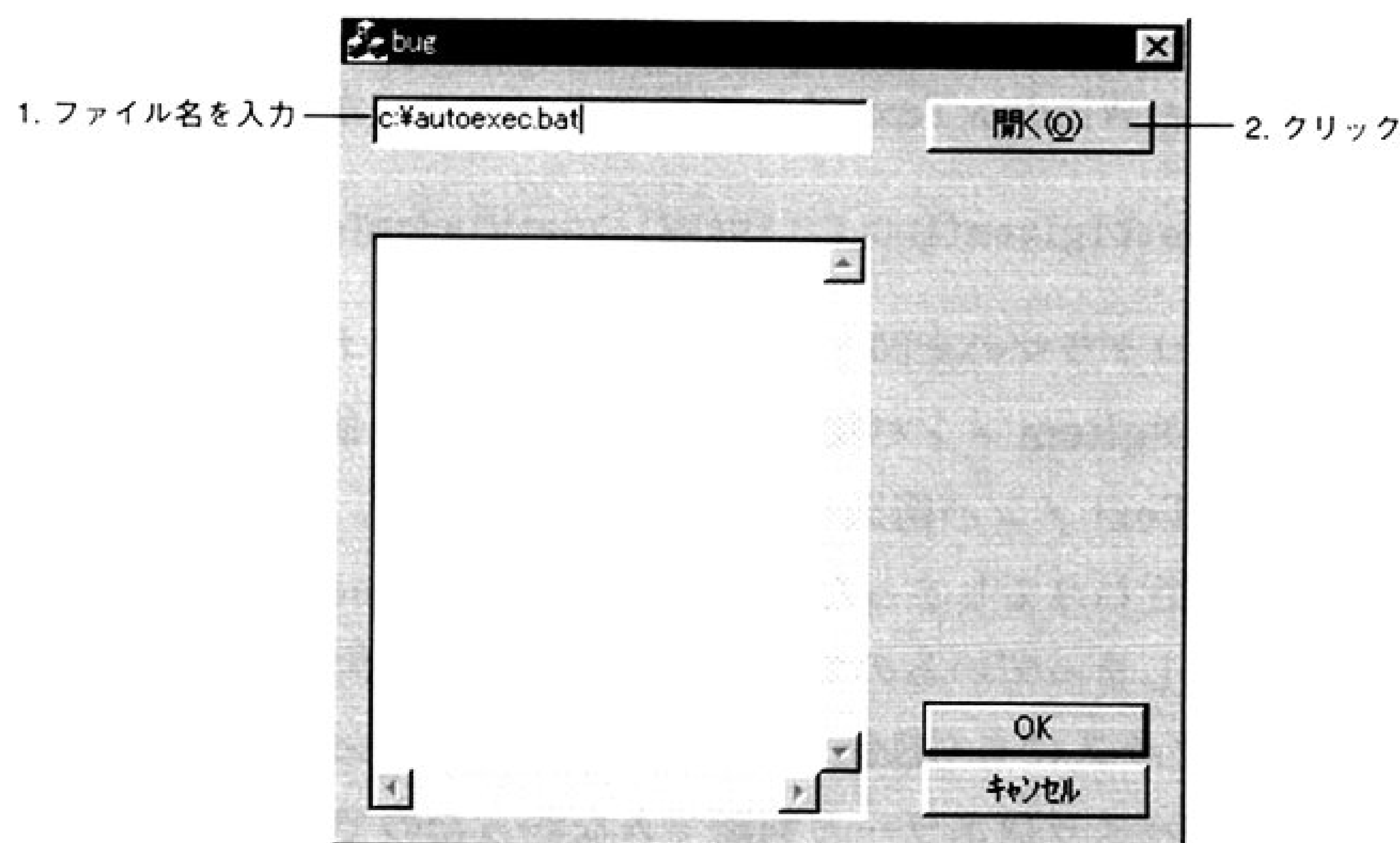


図5-8　Bugダイアログボックス

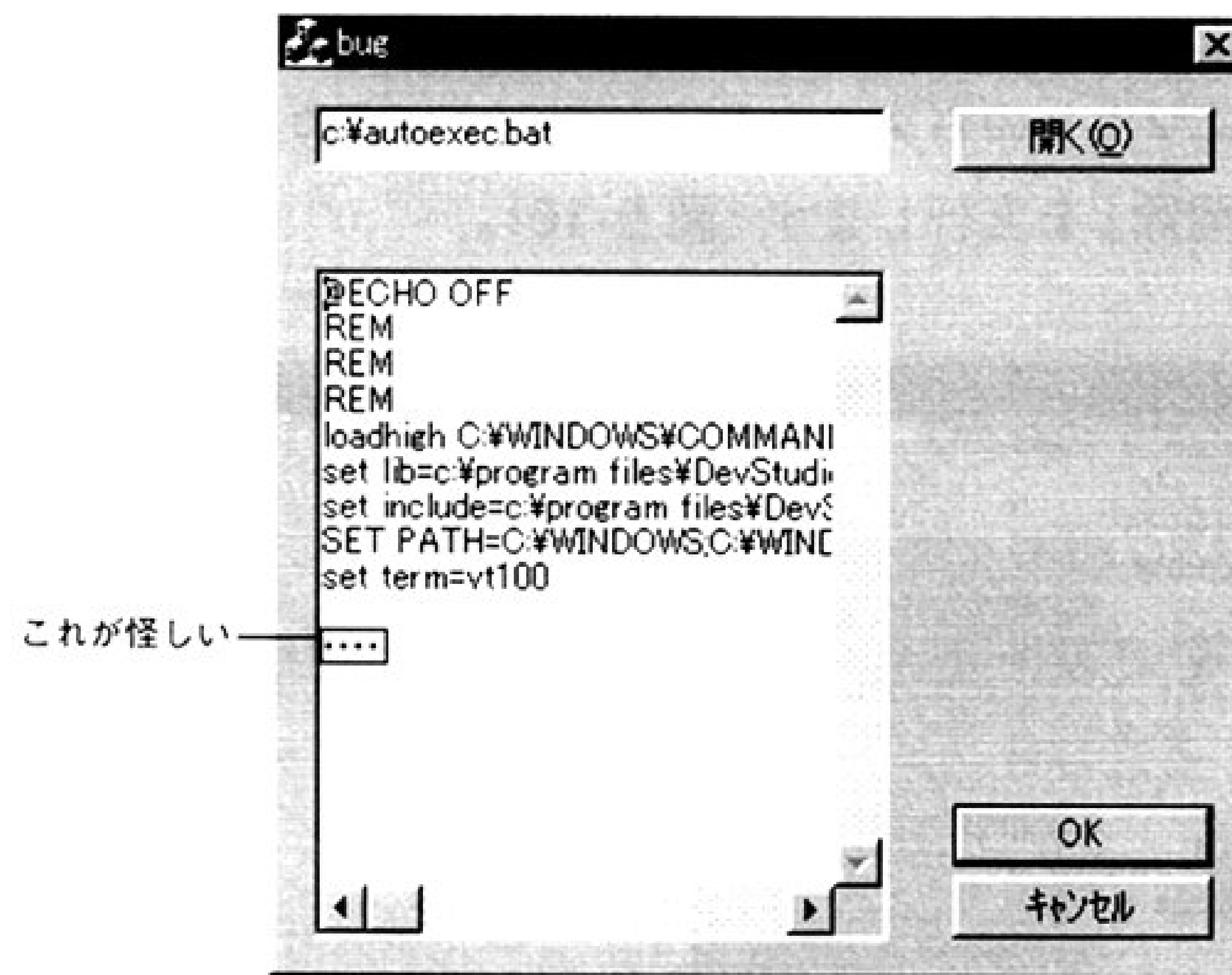


図 5-9 おかしな黒点

で完成、といきたいところですが、表示されたファイルの内容をよく見てください。ファイルの末尾が表示されるようにスクロールさせると、最後にいくつかおかしな黒点が表示されています。

どうやらエディットボックスにゴミが混じり込んでしまっているようです。これは明らかにバグですから調査が必要です。ここはひとまずプログラムの実行を終了し、調査することにしましょう。デバッガで起動したプログラムを終了させれば、自動的にデバッガも終了します。

このような一見動作しているものの不審な動作をする類のバグの調査にはデバッガのブレークポイント機能が便利です。ソースコード上にブレークポイントを設定しておくと、プログラムの実行がブレークポイント位置にさしかかると、実行が自動的に停止され、デバッガに操作が移ります。ここでは CBugDlg::OnOpen メンバ関数の中にバグの原因があることはほぼ間違いないので(何しろここにしかコードを追加していないのですから)、CBugDlg::OnOpen メンバ関数の先頭にブレークポイントを設定し、ここから調査を進めることにしましょう。

リスト 5-2 ブレークポイントの設定

```
void CBugDlg::OnOpen()
{
  CFile f;  // ここにブレークポイントを設定
  CString filename;

  ((CEdit*)GetDlgItem(IDC_FILENAME))->GetWindowText(filename);
  ...
}
```

ブレークポイントを設定するためには、まずブレークポイントを設定したい行にカーソルを移動します。次にマウスの右クリックによりショートカットメニューを開き、[ブレークポイントの挿入／削除] を実行します (**図 5-10**)。



図 5-10　ブレークポイントの設定

するとブレークポイントを設定した行の先頭に赤い●が表示され、ブレークポイントが設定されたことを表します。また設定したブレークポイントは [編集] - [ブレークポイント] を実行すると表示される [ブレークポイント] ダイアログボックスからも確認することができます (**図 5-11**)。このダイアログボックスを使うと、条件付きブレークポイントなどのより高度な機能を利用することができますが、詳細はマニュアルに譲ることにします。

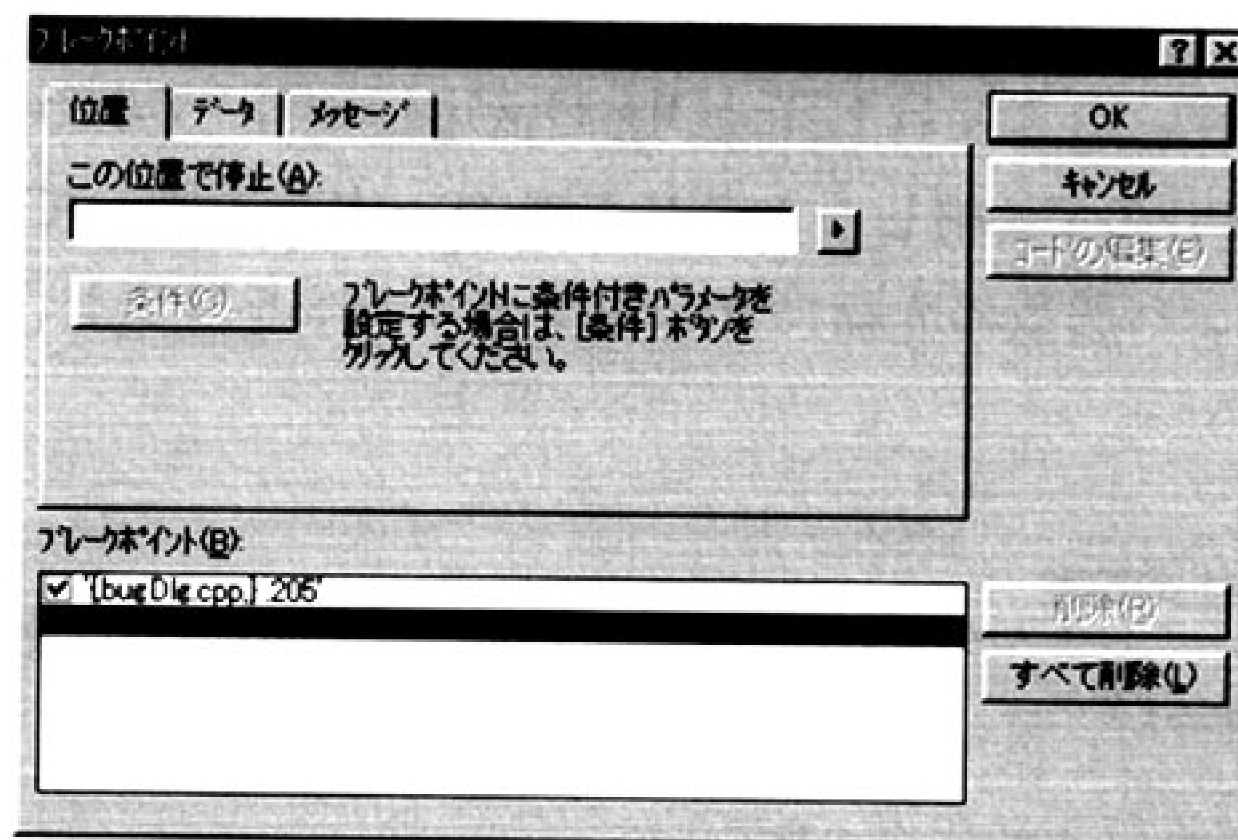


図 5-11　[ブレークポイント] ダイアログボックス

ブレークポイントを設定したら、先と同じようにデバッガを使ってプログラムを実行してください。そして同じ操作を行います。すると今度はファイルの内容が表示される前に実

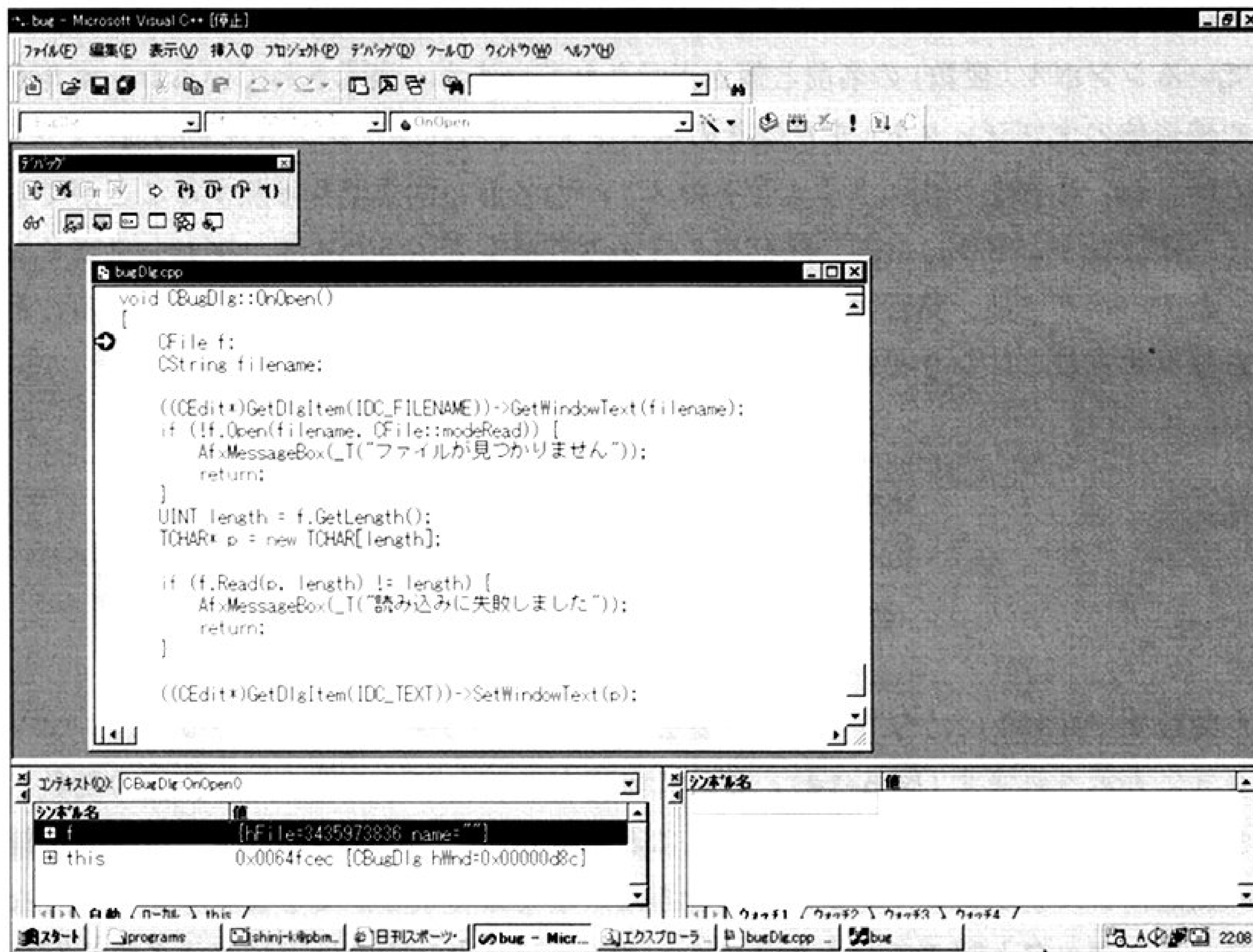


図 5-12　プログラムが停止した

行が停止され、Developer Studio がアクティブになります。このとき黄色い矢印がブレークポイントを設定したソースコード行の先頭に表示されますが、これはその行で実行が停止していることを表しています（**図 5-12**）。また画面には変数ウィンドウやデバッグツールバーといった普段表示されていなかったウィンドウが表示されます（**図 5-13**）。もし変数ウィンドウが表示されなければ、メニューから［表示］－［デバッグウィンドウ］－［変数］を実行してください。



図 5-13　変数ウィンドウ

変数ウィンドウには現在のプログラムカウンタ位置（黄色い矢印が指している位置）で使われているシンボル（変数）の名前と値が表示されています。表示されているシンボルがクラスや構造体のオブジェクト（またはそのポインタ）であれば、シンボル名の前に「＋」記号が表示され、これをクリックするとそのメンバの名前と値までも参照することができます。また変数ウィンドウには関数呼び出し後の返り値も表示されます。ブレークポイントを使ったデバッグでは、実行中に変数の値がどのように変化したのかを追いかけることでバグを発見することになりますから、もっとも重要なウィンドウです。

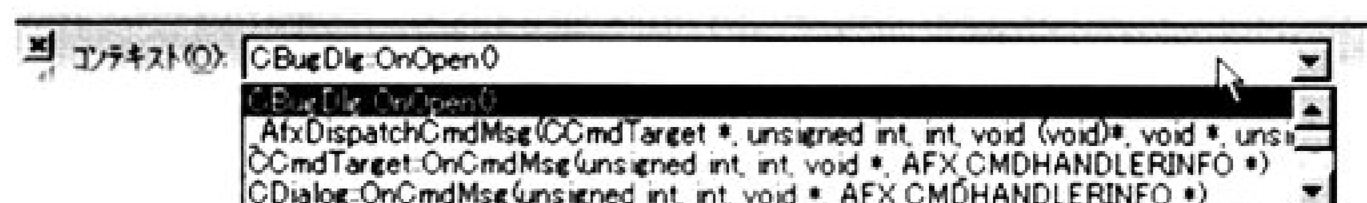


図 5-14 OnOpen 関数のコールスタック

また変数ウィンドウにはもう１つ、重要な情報が［コンテキスト］ドロップダウンコンボボックスに表示されます（**図 5-14**）。ここには現在のプログラムカウンタ位置の関数名が表示されていますが、ドロップダウンすることによって、どのような関数呼び出しを経て現在の位置にプログラムカウンタが進んできたのかを知ることができます。このような情報を**コールスタック**と呼ぶこともあります。ドロップダウン内では下に表示される関数が上に表示される関数を呼び出したことを表しています。ある関数を呼び出すコードが１か所だけであればあまり重要な情報ではありませんが、複数の場所から呼び出される場合には、コールスタックは、どのような条件でバグが発覚するのかを特定するための有益な情報となることが少なくありません。

さて、このままプログラムを停止しておいてもバグを発見できないので、ここからステップ実行を行います。ステップ実行とはソースコードを１行実行するたびに実行を停止する機能です。ステップ実行を一回実行するとプログラムカウンタが１行進み、変数ウィンドウの内容が変化します。このとき値が書き換えられた変数があれば、赤い文字で表示されるようになります。ところでステップ実行には次にあげる３種類が存在します。通常はステップオーバーを実行し、必要に応じてステップイン、ステップアウトを実行するという使い方をします。これらの操作はデバッグツールバーから行います（**図 5-15**）。

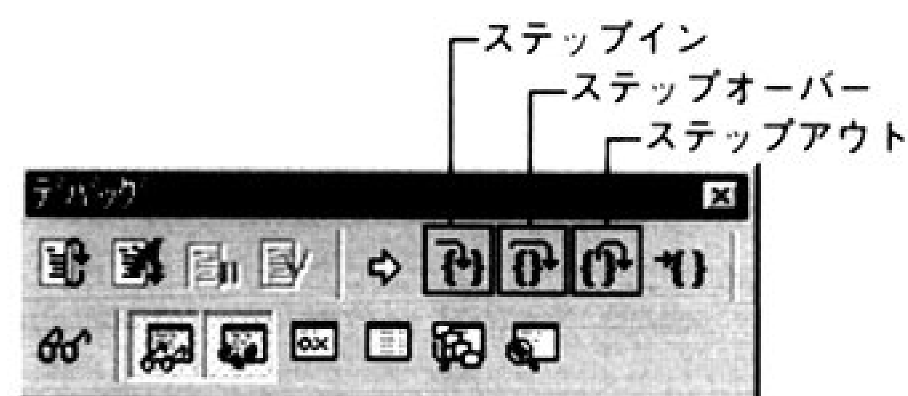


図 5-15 デバッグツールバー

- **ステップイン**
  プログラムカウンタ位置に関数呼び出しがあれば、その関数内にプログラムカウンタを進める
- **ステップオーバー**
  プログラムカウンタを1行進める（関数呼び出しがあれば、その関数の処理が終了した時点までプログラムカウンタを進める）
- **ステップアウト**
  プログラムカウンタ位置の関数が終了するところまでプログラムカウンタを進める

それではブレークポイント位置からステップオーバーを7回実行し、CFile::Read関数の呼び出しが終了した時点までプログラムカウンタを進めてください。すると指定したファイルがメモリに読み込まれますから、これを調査することにします。このときファイルの内容はポインタ変数pが指し示すアドレスに読み込まれていますが、残念ながら変数ウィンドウの変数pの欄には、変数pが指し示すメモリの内容は数十バイト分しか表示されていません。そこでもう1つのデバッグ用ウィンドウであるメモリウィンドウを使ってメモリの内容を調査することにします。メモリウィンドウを使えば、指定したアドレス周辺のメモリの内容を広範囲にわたって参照することができます。つまりこれを使えば読み込まれたファイルの末尾にどのようなデータが含まれているのかを調べることができるというわけです。メモリウィンドウを表示させるためには、［表示］－［デバッグウィンドウ］－［メモリ］を選択します。

メモリウィンドウには、［アドレス］エディットボックスに入力されたアドレスの周辺のメモリの内容が表示されます。［アドレス］には16進数で直接アドレスを指定するだけで



図5-16　メモリウィンドウ

なく、変数ウィンドウに表示されている変数を含めた式を指定することもできます。たとえばポインタ変数pが参照しているメモリの内容を表示させたいときには「*p」と入力します。アスタリスクを忘れないでください。単に「p」と指定すると、ポインタ変数pの値が納められているアドレス(&pで得られるアドレス)が入力されたものとされてしまいます。なお[アドレス]ボックスにキーボードでアドレスを入力する代わりに、変数ウィンドウからシンボル名、またはその値(ポインタ変数の場合)をメモリウィンドウにドラッグ&ドロップすることもできます。

ところで、[アドレス]に変数を入力した場合、その変数の値(アドレス)を簡単に知る方法がなければ、指定したアドレスの内容がダンプリストのどの位置に表示されているのか判断できなくなってしまいます。そんなときは、ダンプリストの適当な位置で一回クリックします。するとダンプリスト上にキャレット(キーボードからの入力用カーソル(縦棒))が表示されます。この状態でアドレスをあらためて指定すると、キャレット位置の行頭に指定したアドレスが表示されます。ダンプリストをスクロールさせて位置を見失ってしまったときなども、こうするとよいでしょう。

ここではファイルの末尾の状態を知りたいのですから、[アドレス]に「*(p+length)」と入力します(lengthはファイルの長さを示す変数)。こうして表示されたメモリの内容と、指定したファイルの内容を比べてみると、どうやらファイルは正常に読み込まれているらしいということがわかります。ということは、p+length以降のメモリの内容までエディットボックスにゴミとして入力されてしまったということです。

そこまでわかったところでソースコードに戻ってみましょう。ここまで追い詰めたらもはや犯人は明らかです。IDC_TEXTエディットボックスにテキストを挿入するために、CEdit::SetWindowTextメンバ関数を使っていますが、このメンバ関数はLPCTSTR型の引数を1つとり、これに指定されたメモリの内容をエディットボックスに挿入します。問題はこの引数が終端に'¥0'を持つ文字列を要求しているところにあります。つまり、ファイルの末尾には'¥0'がありませんから、偶然メモリ上に存在した'¥0'までがファイルサイズを超えてエディットボックスに挿入されてしまっていたわけです。

原因を突き止めればあとは簡単です。次のように修正すればゴミがまじることはなくなります(**リスト5-3**)。

リスト5-3 修正したCBugDlg::OnOpenメンバ関数

```
void CBugDlg::OnOpen()
{
  CFile f;
  CString filename;

  ((CEdit*)GetDlgItem(IDC_FILENAME))->GetWindowText(filename);
```

```
    if (!f.Open(filename, CFile::modeRead)) {
      AfxMessageBox(_T("ファイルが見つかりません"));
      return;
    }

    UINT length = f.GetLength();
    TCHAR* p = new TCHAR[length + 1]; //'¥0'のために+1
    p[length] = '¥0';

    if (f.Read(p, length) != length) {
      AfxMessageBox(_T("読み込みに失敗しました"));
      return;
    }

    ((CEdit*)GetDlgItem(IDC_TEXT))->SetWindowText(p);

    f.Close();
  }
```

これでやっと完成、といいたいところですが、実はまだバグが残っているのです。このサンプルはこのまま次節に持ち越しますので、もうしばらくお付き合いください。

5.2 デバッグサポート関数

前節で紹介したように、デバッガのステップ実行や変数参照といった機能は非常に強力で、大幅にデバッグ効率の向上に貢献してくれます。しかしすべての局面においてデバッガだけでバグを検出できるかというと、そうでもありません。たとえば前章では文字列の末尾を調べるために、メモリウィンドウを使いました。これは変数ウィンドウでは文字列のすべてを表示させることができなかったからです。このようにデバッガが用意している各種のツールにも得手不得手があります。先の例では文字列データを調べればよかったため、メモリウィンドウが利用できましたが、これが数値の配列だったらどうでしょう。さらに、複雑なクラスオブジェクトのリンクリストだったら? 変数ウィンドウにこれを表示させることはできず、またメモリウィンドウで追跡することも容易ではありません。

こうしたデータ構造のデバッグを行うためには、デバッガだけに頼らず、ソースコードの中にデータの正当性を検出するための、デバッグコードを埋め込む手法が有効です。MFCにはこうしたデバッグコードをサポートするクラスや関数が豊富に用意されています。ここでは先のプロジェクトbugを使って、これらのデバッグサポート機能を利用してみます。

メモリリーク

数あるバグの種類の中でも、メモリ管理にかかわるバグ、とくにメモリリークほど厄介なバグはないでしょう。メモリリークが発生しても一見プログラムは正常に動き続けるものの、あるタイミングで突然プログラムが異常終了します。そして異常終了した箇所がデバッガによってレポートされたとしても、メモリリークを起こしている行から遠く離れていることがしばしばです。しかしMFCを使っていれば、メモリリークが発生していることがレポートされるばかりか、メモリリークを引き起こしているメモリ確保の位置まで特定してくれます。

またライブラリにはオーバーランのチェック機構も用意されています。たとえばnew演算子を使って要素数10のchar型変数の配列を確保したとしましょう。この配列の11番目の要素に書き込みを行うことは、当然のことながらプログラミング上のエラーであり、もともとそのメモリに格納されていた内容は破壊されてしまいます。ところがデバッグ用MFCを利用すれば、delete演算子を使ってメモリブロックを解放するとき、オーバーランが発生したことを検出し、これをレポートすることができます。

このようにプロジェクト構成Win32 Debugを利用したときにリンクされるデバッグ用MFCには豊富なデバッグサポート機能が用意されています。とくにメモリ管理に関するサポート機能が充実しています。それだけメモリ管理はプログラマにとって恐ろしいバグの温床なのです。ところで、プロジェクト構成Win32 Releaseを使ってリリース用実行ファイルを作成するときには(正確には_DEBUGが定義されていないとき)、これらのチェック機構はすべて無効になるようになっています。十分テストしたあとにリリース用実行ファイルを作るのですから過剰なチェックは処理速度を低下させるだけであり、必要ないというわけです。

ここではメモリリークに関するMFCのデバッグサポート機能について解説します。

メモリリークの検出

前節の終わりで作ったプロジェクトをビルドし、これを実行してください。一見問題なく動作しているようですが、アウトプットウィンドウを見ると、実はそうでもないことがわかります。プログラムを終了させて、アウトプットウィンドウのデバッグタブを表示させると、次のようなメッセージが表示されているはずです。

```
Detected memory leaks!
Dumping objects ->
C:¥usr¥vcpro¥bug¥bugDlg.cpp(185) : {58} normal block at 0x00771780,
676 bytes long.
 Data: <C:¥PROGRA~1¥     > 43 3A 5C 50 52 4F 47 52 41 7E 31 5C B3 B2 D9 BD
Object dump complete.
```

このメッセージは「メモリリークが発生した」ことを伝えています。メモリリークとはどこからも参照されていないにもかかわらず、解放されずに放置されているメモリが存在しているということです。C++言語ではnew演算子を使ってメモリを確保した場合、使用済みになった時点でdelete演算子を使ってそのメモリを解放しなければなりません。そのメモリの解放を怠った場合、無駄にメモリが消費されることになりますが、これがメモリリークです。実際には十分なメモリが装備された計算機でアプリケーションを動かしている限り、メモリリークが発生してもアプリケーションの動作そのものには支障のないこともあります。しかしメモリリークはえてしてメモリ管理のミスから発生するものであり、わかりにくい、なんらかのバグの副作用として起こることも多いのです。これは決して見逃してはならないメッセージです。

では、先のエラーメッセージを詳しく説明しましょう。1つのメモリリークは2行に渡ってレポートされます。つまりここでは1つのメモリリークが発生しているわけです。まず1行目ですが、次のフォーマットで出力されています。

```
<ソースファイル名>(<行番号>):{<リクエスト番号>}<オブジェクト種> at <アドレス>,
    <サイズ> bytes long
```

つまり、先の例ではソースファイルC:¥usr¥vcpro¥bug¥bugDlg.cppの185行目で確保したメモリが解放されていないことがレポートされているわけです。＜リクエスト番号＞は、アプリケーションの開始から何回目のメモリ確保を行った時点でのメモリリークかを表しています。先の例では58回目に確保されたメモリであることを表しています。また＜オブジェクト種＞としてnormal blockがレポートされていますから、これはクラスのオブジェクトではなく、単純な型として確保されていることを表しています。 さらにそのメモリはアドレス0x00771780から確保され、サイズは676バイトであることがわかります。

2行目にはリークしているメモリの内容の先頭16バイト分が文字と16進数でダンプされています。両者は同じデータを、形を変えて表示しているだけです。メモリの内容が文字列であればキャラクタ表示が有益ですし、バイナリデータであれば16進数ダンプが役に立ちます。先の例では文字列の保存に使ったメモリがリークしていたため、キャラクタ表示を見ればどういった目的で使われたメモリであるか推測することができます。

以上の情報から、メモリリークを解消することにしましょう。この場合、レポートされたソースコード行におそらくnew演算子があり、それに対応するdelete演算子が存在しないであろうことが予想できますから、ともかくソースファイルを表示しましょう。このためには、ビルドエラーが起きた行を表示させたときと同じように、やはり F4 を使います。このようにアウトプットウィンドウに表示されたレポートに対しては、すべて F4 でジャンプが可能となっています(ファイル名と行番号が行頭に表示されている場合)。

エラー行を表示させると確かに次のようなnew演算子があり、これに対応するdelete演算子は存在しません。確保したメモリへのポインタはローカル変数pに保存されるため、

CBugDlg::OnOpen メンバ関数が終了した時点でメモリリークが発生してしまうわけです。

```
TCHAR* p = new TCHAR[length + 1]; //'¥0'のために+1
```

そこで**リスト 5-4** に示すように CBugDlg::OnOpen メンバ関数を修正します。

リスト 5-4　メモリリーク解消版 CBugDlg::OnOpen メンバ関数

```
void CBugDlg::OnOpen()
{
  CFile f;
  CString filename;

  ((CEdit*)GetDlgItem(IDC_FILENAME))->GetWindowText(filename);
  if (!f.Open(filename, CFile::modeRead)) {
    AfxMessageBox(_T("ファイルが見つかりません"));
    return;
  }

  UINT length = f.GetLength();
  TCHAR* p = new TCHAR[length + 1]; //'¥0'のために+1
  p[length] = '¥0';

  if (f.Read(p, length) != length) {
    AfxMessageBox(_T("読み込みに失敗しました"));
    return;
  }

  ((CEdit*)GetDlgItem(IDC_TEXT))->SetWindowText(p);

  delete [] p;  //確保したメモリは解放する

  f.Close();
}
```

メモリリークの瞬間

ところで、この例ではエラーの修正は容易でしたが、エラー行がわかっただけでは修正が難しい場合もあります。たとえばループの中で new 演算子を繰り返し呼び出している場合や、条件分岐によって new 演算子が実行されたりされなかったりする場合では、どのような条件で new 演算子が実行されたときにメモリリークが発生するのかを突き止めなければなりません。このようなときに便利なユーティリティ関数が AfxSetAllocStop 関数です。

void AfxSetAllocStop(LONG *lRequestNumber*)

引数　**LONG** *lRequestNumber*　　リクエスト番号を指定する

この関数の使い方は簡単です。この関数は、ほとんどの方が「何に使うのだろう？」と思

われたはずの、メモリリークメッセージのリクエスト番号({}の間に表示される番号です)を使います。手順としては、まず一度アプリケーションを実行してメモリリークを発生させ、アウトプットウィンドウに表示されるリクエスト番号を調べます。次にメモリリークが発生しているnew演算子やmalloc関数などの直前に、調べたリクエスト番号を引数に指定したAfxSetAllocStop関数の呼び出しを挿入します。たとえばリクエスト番号58でメモリリークが発生しているならば、

```
AfxSetAllocStop(58);
```

という行を挿入します。ソースコードの修正が済んだらビルドし、あらためてデバッガから起動します。すると指定したリクエスト番号に対応するメモリの確保が行われた瞬間に、あたかもブレークポイントを見つけたかのように実行を停止して、デバッガで調査を開始できます。これを使えば、メモリリークが発生した瞬間の変数の値やコールスタックを調べることができます。

デバッグが終わったら、AfxSetAllocStop関数を含む行は削ってしまって構いません。

スナップショット

MFCによるメモリリークの検出はアプリケーションが終了したときに行われます。アプリケーションが終了するまではすべてのメモリが参照される可能性があるのですから、これも当然のことです。しかしそれだけではメモリリークの発生箇所を特定するのが困難な場合もあります。たとえばループの中でメモリを確保している場合に、2回目以降でメモリリークが発生していたり、特定のオプションをチェックしてダイアログボックスを閉じたときにだけリークしていたり、条件がそろったときにだけメモリリークが発生していても、こうした条件はメッセージから読み取ることはできません。これを補う関数がAfxSetAllocStop関数だったわけです。しかしこの関数は完全に再現性のある場合にしか役に立ちません。実行するたびに、メモリリークを発生するリクエスト番号が変化するような場合には、リークの瞬間を捕らえることはできないのです。

先にアプリケーションが終了するまでメモリリークとは判断できないと述べましたが、それはライブラリが判断できないだけで、プログラマにとってはそうではありません。プログラマはソースコードを読むことで、特定のエリアでは完全にnew演算子とdelete演算子の対応が閉じていることを判断できます。つまりnewとdeleteが閉じているはずの、特定のソースコード行のエリアを設定し、このエリアに入ったときのメモリブロックの数と、エリアを出たときのメモリブロックの数を調べて、これが一致していなければメモリリークが発生していると判断できるわけです。このように、ある時点でのメモリの状態(スナップショット)をとってメモリリークを検出するためのクラスがCMemoryStateクラスです。CMemoryStateクラスの主なメンバ関数を**表5-3**に示します。

メンバ関数	処理
void CMemoryState::Checkpoint()	スナップショットをとる
BOOL CMemoryState::Difference (const CMemoryState& oldState, const CMemoryState& newState);	2つのスナップショットから差分を作る
void CMemoryState:: DumpStatistics()	スナップショットの内容を表示する

表 5-3　CMemoryState クラスのメンバ関数

この3つのメンバ関数は次のように組み合わせて使います。

```
CMemoryState begin, end, diff;
begin.Checkpointo();  // この時点でのメモリの確保状況のスナップショットを begin にとる
...
// この範囲では new と delete の関係は閉じているものとする
...
end.Checkpoint(); //  この時点でのスナップショットを end にとる
if (diff.Difference.(begin, end)) {
  // CMemoryState::Difference は begin と end の差分が空でなければ TRUE を返す
  diff.DumpStatistics();  // 差分を表示する
}
```

このようなコードを埋め込んでおけば、メモリリークが発生すると CMemoryState::Dump Statistics メンバ関数によって差分の状況が次のようにアウトプットウィンドウに出力されます。これによって、いくつのメモリブロックがリークしているのか、そしてそのサイズはどれほどであるのかを知ることができます。

```
0 bytes in 0 Free Blocks.
1100 bytes in 1 Normal Blocks.
0 bytes in 0 CRT Blocks.
0 bytes in 0 Ignore Blocks.
0 bytes in 0 Client Blocks.
Largest number used: 1054 bytes.
Total allocations: 1399 bytes.
```

この例では Normal Block が1つリークしており、そのサイズは 1100 であることを意味しています。

差分は5種類のブロックタイプに分類されてレポートされます。それぞれのブロックタイプの意味は**表 5-4** に示すとおりです。

表 5-4 の中で「Free Blocks」だけが異彩を放っています。なぜならここにレポートされるメモリブロックは「解放されていないブロック」なのですから、普通ならば「Free Blocks（解放済みブロック）」がリストアップされるはずはないからです。それでもここにそのような項目が用意されているのは、MFC に解放したメモリブロックを保存しておく機能があるからです。ただしこの機能はデフォルトでは無効になっています。この機能を有効にする

ブロックタイプ	目的
Free Blocks	解放されたメモリブロック（後述）
Normal Blocks	char や int といった基本型や CObject クラスを継承していないクラスのために確保されたメモリブロック
CRT Blocks	C ランタイムライブラリ内で確保されたメモリブロック
Ignore Blocks	メモリリーク検出機構をオフにしているときに確保されたメモリブロック
Client Blocks	CObject クラスおよびその派生クラスのオブジェクトのために確保されたメモリブロック

表 5-4　ブロックタイプ

ためには、MFC 内部で定義されているグローバル変数 afxMemDF に delayFreeMemDF を設定する必要があります。具体的には次のようなコードが実行された行以降から、機能するようになります。論理和で代入しているのは、afxMemDF にはその他のメモリ管理用フラグも設定されているからです。

```
afxMemDF |= delayFreeMemDF;
```

この機能を利用すると、delete 演算子や free 関数によってメモリを解放しても、そのメモリブロックは再利用されなくなります。つまり解放したときのデータがそのままメモリに残るため、割り当てられていたメモリの内容のチェックが可能になります。ただし、その副作用としてシステムがメモリ不足状態になりやすくなります。通常解放されたメモリは次のメモリ確保要求によって再利用されますが、これがまったく行われなくなるため、しばらくするとメモリ不足状態になるわけです。これを利用して、意図的にメモリにストレスをかけるテストに用いることもできます。

● 診断関数

デバッガにしろ、メモリリーク検出機能にしろ、ここまでのデバッグ技法はすべて、バグが発覚してからの対処に使うものでした。確かにデバッグはバグが発覚してから行うものですが、あらかじめバグが発覚したときに備えておくことも可能です。

1 つは要所要所に実行時の正当性を確かめるチェックポイントを設けておく方法です。たとえばメモリアクセスに失敗してプログラムが異常終了するのを待つよりは、ポインタの計算が済んだ時点でポインタが指し示すメモリに期待どおりのデータが存在しているのかをチェックすれば、ポインタの計算に問題があることがたちどころにわかります。

診断のタイミングとしては、関数が呼び出された直後などが考えられます。たとえば列挙型（enum）の値を受け取る関数では、列挙型で定義された整数値の範囲だけが正当な値となります。これを if 文でチェックすることもできますが、正しく書かれたコードならば、範囲外の値が渡されることはないはずです。これを if 文で毎度チェックするのは時間の無

駄というものです。それならばデバッグ環境でのみ診断コードが実行されるようにしておくのが得策です。

もう1つはオブジェクトのダンプを容易にする方法です。printf文などによる変数の値の出力は簡単に行えることから、よく用いられるデバッグ技法ですが、クラスや構造体のオブジェクトに関してはそうもいきません。すべてのメンバ変数を読みやすい形で出力させるためには何行ものコードが必要です。さらにメンバ変数が変更になって、増減したり、型が変わったり、有効な値の範囲が変わったりということはプログラムの開発中に頻繁に起こることです。これに対応してダンプコードを変更してまわる作業は大変な労力が必要です。そこでMFCにはオブジェクトを簡単にテキスト形式で出力するための標準的な方法が用意されています。

以下ではこの2点について解説します。

式の診断（ASSERTマクロ、VERIFYマクロ）

もっとも基本的な診断マクロがASSERTマクロです。この関数はC言語のassert関数によく似ていますが、デバッグ環境でのみ機能するところが違います。つまりWin32 Debugプロジェクト構成で定義されている_DEBUGマクロがなければASSERTマクロは空に定義されるので、その行はコンパイルされません。いちいちASSERTマクロを#ifdef～#endifでくくったり、削って回ったりしなくても、リリース時にはASSERTマクロに関連する余分なコードサイズや処理時間を省くことができます。

ASSERTマクロの使い方は次のとおりです。

ASSERT(BOOL *expression*)

引数　**BOOL** *expression*　　正しく動作している限り真となる条件式を指定する

たとえば以下のようなASSERTマクロがあったとします。

```
ASSERT(p != NULL)
```

これはプログラマによる「この位置でpがNULLになることはありえない」という主張です。もしpがNULLであれば、つまりASSERTマクロの引数に指定した式が成り立たなかった場合、**図5-17**に示すようなダイアログボックスが表示されます。ここには＜中止＞、＜再試行＞、＜無視＞の3つのボタンが並んでいます。＜中止＞ボタンをクリックすれば、プログラムの実行が終了されます。＜再試行＞ボタンをクリックすると、デバッガが起動され、アサーションに失敗した箇所からデバッグを開始できます。＜無視＞ボタンをクリックすると、ASSERTマクロの次の行から実行を続けます。

ASSERTマクロはMFCの内部でたくさん使われているので、自分のソースコードに1つも記述していなくてもアサーションに失敗してプログラムが終了することがあります。この場合、アサーションに失敗した関数を呼び出す手前になんらかのバグがあったと考えら



図5-17 Assertion Failed

れます。ASSERTマクロは主に関数が呼び出された直後に、引数が正当なものであるかをチェックする目的で使われるため、正しく引数を渡しているかを確かめるとよいでしょう。

ところでASSERTマクロを使うときにはexpressionに副作用のある式を記述してはいけません。なぜならASSERTマクロはデバッグ環境だけでしか機能しないので、リリース環境とデバッグ環境でプログラムの動作が変わってしまうからです。たとえば次のような式を指定すると、リリース環境とデバッグ環境ではiの値が異なってきます。

```
ASSERT(i++ != 0)
```

どうしてもこのような指定がしたければ、ASSERTマクロの兄弟マクロであるVERIFYマクロを使います。このマクロはリリース環境でもデバッグ環境でも式の評価を行うASSERTマクロと考えられます。ただしリリース環境では式の値が真であるかどうかのチェックは行われません。ASSERTマクロとVERIFYマクロによるマクロ展開後の違いは**表5-5**に示すとおりです。

	ASSERT(i++ != 0)	VERIFY(i++ != 0)
デバッグ環境	assert(i++ != 0)	assert(i++ != 0)
リリース環境	なし	i++ != 0

表5-5 ASSERTマクロとVERIFYマクロの違い

クラスオブジェクトの診断(CObject::AssertValidメンバ関数)

ASSERTマクロは単純な式の評価には便利なマクロですが、複雑な式を評価する場合にはいくつもの式を&&で結んだりすることにより、ソースコードがわかりにくくなることがあります。とくにクラスオブジェクトの診断をしたければ、すべてのメンバ変数に関しての診断をしなければなりませんし、ASSERTマクロを使った診断用のコードを何か所にも埋め込むのは不経済です。それにメンバ変数はprivateかprotectedである場合が多いので、こうした場合にはクラスの外からメンバ変数に対して診断を行うことはできません。

そこでMFCにはクラスのオブジェクト専用バージョンのASSERTマクロ、ASSERT_

VALIDマクロが用意されています。ASSERT_VALIDマクロを使うと、クラスオブジェクトへのポインタを渡すとそのオブジェクトについて診断を行い、問題があればASSERTマクロと同じようにダイアログボックスを表示してアサーションの失敗をレポートすることができます。そのしくみは、目的のクラスにAssertValidメンバ関数を実装しておき、ASSERT_VALIDマクロによってこれを呼び出すというものです。ただしAssertValidメンバ関数はCObjectクラスの仮想メンバ関数として呼び出されるため、目的のクラスはCObjectクラスの派生クラスである必要があります。

CObject::AssertValidメンバ関数のプロトタイプ宣言は次のとおりです。これにしたがってオーバーライドし、その中でASSERTマクロなどを使って診断を行います。

virtual void CObject::AssertValid( ) const;

たとえば、CFile*型メンバ変数fとUINT型メンバ変数lenを持つCAssertTestクラスが次のように定義されているとします。ここでのポイントはCObjectクラスの派生クラスであること、もう1つはAssertValidメンバ関数の定義を#ifdef _DEBUG～#endifで囲むことです。CObject::AssertValidメンバ関数は他のデバッグ関数群と同じく_DEBUGマクロが定義されているときのみ定義されるので、この処置は必須です。

```
class CAssertTest : public CObject  // CObject クラスの直接、
                                    // または間接派生クラスを作る
{
protected:
  CFile* f;
  UINT len;
public:
  ....
#ifdef  _DEBUG
// AssertValid メンバ関数はデバッグ環境でのみ定義されるので、
// #ifdef で囲む必要がある
  virtual void AssertValid() const;
#endif
};
```

このときのCAssertTest::AssertValidメンバ関数の実装例は次のようになります。ここでのポイントは、基底クラスであるCObjectクラスのAssertValidメンバ関数を呼び出しているところです。CObject::AssertValidメンバ関数にはCObjectクラスに対する診断コードがあるわけですが、CAssertTestクラスによってオーバーライドされてしまうとこれが機能しなくなってしまいます。そこで明示的に基底クラスのメンバ関数を呼び出しているのです。次にCAssertTestクラスのメンバ変数の診断を行います。まずCFile*型オブジェクトfですが、CFileクラスにはMFCによってAssertValidメンバ関数が実装されているので、ASSERT_VALIDマクロを使って診断を依頼することができます。このように各クラスごとに診断コードを用意しておくことで、1つのクラスですべての診断を行う

必要がなくなります。なお MFC で定義されているクラスのほとんどには AssertValid メンバ関数が実装されています（ただし CObject クラスの派生クラスに限る）。最後に基本型 UINT のメンバ変数 len に ASSERT マクロを使って診断は終わりです。

```
#ifdef  _DEBUG  // 関数の実装全体を#ifdef～#endif で囲む
void CAssertTest::AssertValid() const
{
  CObject::AssertValid();
  // 基底クラスである CObject クラスの AssertValid メンバ関数を呼び出す
  if (f != NULL) {
  ASSERT_VALID(f);
  // CFile クラスオブジェクト f を診断するために
  // CFile::AssertValid メンバ関数を呼び出す
  ASSERT(f->GetLength() == len);
  // オブジェクト f が指すファイルの長さと
  // メンバ変数 len の値が同じであることを確かめる
  }
 }
#endif
```

こうして実装が済んだら、次のように ASSERT_VALID マクロを使って CAssertTest クラスオブジェクトの診断が可能になります。すると CAssertTest::AssertValid メンバ関数が実行され、そこに含まれるすべての診断を通過したときだけ、CAssertTest クラスの診断が成功に終わります。

```
CAssertTest* p = new CAssertTest;
ASSERT_VALID(p);
```

CObject::Dump メンバ関数

printf 関数や cout を使ってデバッグメッセージを出力させるのは昔から行われているデバッグ手法ですが、クラスオブジェクト全体をデバッグメッセージとして出力させようとすれば、大変な作業です。事情としては AssertValid メンバ関数と同じようなものです。そこで MFC にはクラスオブジェクトをすべて人間が読める形にして出力する標準的な方法が提供されています。それが CObject::Dump メンバ関数です。

CObject::Dump メンバ関数のプロトタイプ宣言は次のとおりです。CObject::AssertValid メンバ関数と同じく CObject クラスの仮想メンバ関数として定義されています。したがって Dump メンバ関数を使うためには、目的のクラスを CObject クラスの派生クラスとする必要があります。

virtual void CObject::Dump(CDumpContext& *dc*) const;

引数にとる CDumpContext クラスは CArchive クラスに似たクラスです（詳しくは第 3 部を参照のこと）。つまり CDumpContext オブジェクトである dc を使えば、

```
int a;
char b[] = "debug message: ";
dc << b << a;
```

のように<<演算子を使ってデバッグメッセージを出力することができます。出力先は通常、アウトプットウィンドウのデバッグタブです。

前節でのCAssertTestクラスと同じように、CDumpTestクラスを次のように定義して、Dumpメンバ関数の実装例を示します。クラス定義でのポイントはAssertValidメンバ関数とまったく同じです。繰り返しになりますが、CObjectクラスの派生クラスであること、#ifdef _DEBUG～#endifでDumpメンバ関数のプロトタイプ宣言を囲むのを忘れないようにしてください。

```
class CDumpTest : public CObject  //CObject クラスの直接、
                                  // または間接派生クラスを作る
{
protected:
  CFile* f;
  UINT len;
public:
  ....
#ifdef  _DEBUG
// Dump メンバ関数はデバッグ環境でのみ定義されるので、
#ifdef で囲む必要がある
  virtual void Dump(CDumpContext& dc) const
#endif
};
```

このときのCDumpTest::Dumpメンバ関数の実装は次のとおりです。ここでも構造はAssertValidメンバ関数とよく似ています。まずCObject::Dumpメンバ関数を明示的に呼び出します。目的はCObjectクラスをダンプすることです。次にCFileクラスへのポインタをdcに出力しています。これによってCFile::Dumpメンバ関数が呼び出され、CFileクラスオブジェクトがダンプされます。最後にUINT型メンバ変数lenを出力します。これでCDumpTestクラスのメンバ変数すべてをダンプする準備が整いました。

```
#ifdef  _DEBUG
void CDumpTest::Dump(CDumpContext& dc) const
{
  CObject::Dump(dc);
  dc << f;
  dc << "file length: " << len;
}
#endif
```

Dump メンバ関数の実装が済んだら、プログラム中のどこからでも次のコードを書き込むことでクラスをまるごとダンプすることができます。

```
CDumpTest* p = new CDumpTest;
#ifdef  _DEBUG
afxDump << p;
#endif
```

afxDump は MFC の内部で定義されている CDumpContext オブジェクトです。ただしデバッグ環境でのみ定義されているので、#ifdef で囲むことを忘れずに。

以上で Visual C＋＋でのデバッグ手法の解説を終わります。Visual C＋＋にはデバッガ、ライブラリの両面からさまざまなデバッグに便利な機能が提供されていますが、伝統的に使われているデバッグ手法も忘れないようにしてください。もっとも効果のあるデバッグ手法は、「関係のないコードを削った、コンパクトなプログラムを作ること」です。まずはプログラムの動作に支障をきたさない範囲でソースコードを削り、怪しい箇所を狭めていくことです。また可能な限りクラスごとに小さな動作チェック用プログラムを作っておくとよいでしょう。

第3部

MFCを使ってみよう

第1部ではDeveloper Studio、AppWizard、リソースエディタ、ClassWizardなどを利用したプログラム作成の手順とMFCを使ったプログラムの基本的な構造について説明しました。第2部では、グラフィックの表示やダイアログボックスなどを使って、実際にMFCを使ったVisual C++プログラミングに簡単に触れてみました。さて、ここからが本番です。第3部ではいよいよフレームワークとしてのMFCの機能を使って、少々本格的なプログラム作成に挑戦してみます。さまざまな要素が複雑に絡み合いながらプログラムが構築されていきますから、クラスの構造やオブジェクトの相互関係に注意し、コードの流れを把握することが大切になります。

1 テキストエディタを作ってみよう

第3部で作成するプログラムは、2種類のデータ、すなわちテキストデータと図形の描画データを扱います。つまり、テキストエディタ+ドローツールという構成のプログラムを作ることが第3部の目標です。まず、最初にMFCに用意されている便利なクラスを使って、テキストエディタを作成することにします。しかし、その前に第1部で若干の説明をしたドキュメントとビューについて、もう一度簡単におさらいしておきましょう。なぜなら、ここからはドキュメントとビューというMFCを支える2つの重要なクラスを利用してプログラムを構築していくからです。ドキュメントとビューの関係についての説明が終わったら、プロジェクト全体の設計をして、次にテキストエディタの作成に取りかかります。

1.1 残してこそ意味のあるドキュメント

Visual C++でいう**ドキュメント**とは、アプリケーションが扱うデータ全般を指す言葉です。たとえば、Aというアプリケーションはテキストというドキュメントを扱い、Bというアプリケーションはドローデータというドキュメントを扱う、というように各アプリケーションにはそれぞれ扱うべきドキュメントの種類があります。これを本書では**ドキュメントタイプ**と呼びます。このように一般的に使われるドキュメントという言葉よりも広い意味を持たせていることに注意してください。また**ビュー**とは、ユーザーとアプリケーションとの間のインターフェイスを指す言葉です。つまり、ユーザーがアプリケーションに対して指示を出す方法と、アプリケーションがユーザーに対して情報を示す方法を合わせてビューと呼びます。そしてVisual C++とMFCを使うときには、このドキュメントとビューを明確に分離してプログラミングすることが要求されるということを第1部では述べました。

ところが第1部で触れて以来、ドキュメントやビューにはとくに注意をはらわずに済ませてきました。しかし第3部ではビューだけでなくドキュメントの実装も行います。簡単

にいってしまえば、「アプリケーションが作成したデータをファイルに保存する方法を示す」ということです。いよいよ MFC に用意されたフレームワークとしての機能が本領を発揮し始めます。

●ドキュメントとビュー

Visual C++では、ドキュメントとビューという概念を用いたプログラミングスタイルを指して、**ドキュメント－ビュー・アーキテクチャ**と呼んでいます。それでは実際にドキュメント－ビュー・アーキテクチャを用いてプログラムを作成するためには、どういったコードを記述すればよいのでしょう？

ドキュメントとビューをそれぞれ別々のクラスを使って実装するというのが、ドキュメント－ビュー・アーキテクチャの基本姿勢だというのは第1部でもお話ししたとおりです。

まず、ドキュメントクラスとして CDocument クラスの派生クラスが必要です。派生クラスを作るわけですから、ドキュメントを管理するために必要な機能は CDocument クラスが提供してくれます。したがって、派生クラスには、ドキュメントを保存するための変数を追加したり、追加したメンバ変数を操作するメンバ関数を追加するだけで、ドキュメントクラスとして必要な機能は用意できます。プログラマが記述しなければならないこれらのメンバ関数の中でもっとも重要なのが Serialize 関数です。この関数はメモリ上のドキュメントをファイルに保存したり、ファイル上のドキュメントをメモリに読み込んだり、ドキュメントクラスの中心となる関数です。

また、ビュークラスとして、CView クラスの派生クラスも必要です。ビュークラスはすでに何度も扱っていますから、その動作はよくわかっていることと思いますが、マウスやキーボードを使ったユーザーからの入力を受け取ったり、ウィンドウに文字や図形を表示するのに使います。しかし、ドキュメントを扱いはじめるとそれに加えて、ユーザーのビューへの操作をドキュメントに反映しなければなりませんし、ドキュメントの内容が変更されたら、それをウィンドウなどに反映しなければなりません。このような処理を行うには、ドキュメントクラスとの連携プレイが重要になってきます。今までのようにユーザーの操作をすぐさまウィンドウに反映させるのではなく、一度ビュークラスからドキュメントクラスにユーザーの操作を伝え、ドキュメントに変更を加えてから、改めて変更されたドキュメントの内容をウィンドウなどに反映させる、といった処理を行うことも必要になってきます。

アクセス元	取得するオブジェクト	メンバ関数
ドキュメントクラス	ビュークラスのオブジェクト	GetFirstViewPosition/GetNextView
ビュークラス	ドキュメントクラスのオブジェクト	GetDocument

表 1-1 他方のクラスのオブジェクトを取得するメンバ関数

このようにビュークラスとドキュメントクラスは別々のクラスではあっても、密接にかかわっているため、互いのメンバにアクセスすることがよくあります。そこで、CDocumentクラスとCViewクラスには他方のクラスのオブジェクトを取得するためのメンバ関数が用意されています（**表1-1**）。これらの関数を使えば、他方のクラスのパブリックなメンバにアクセスできます。

1.2 プロジェクトの設計

第3部で作成するのは、テキストエディタ＋ドローツールですが、まずはおおざっぱにプログラムの構成をつかむために、その全体的な構成を考えてみましょう。第1部でも述べましたが、1つのドキュメントタイプにつき、1つのドキュメントクラスとビュークラスが必要なことを覚えているでしょうか（リソースとフレームウィンドウも必要だがここでは2つのクラスに注目する）。ここでは2つのドキュメントタイプを扱うのですから、2組のドキュメントとビューが必要になります（**図1-1**）。

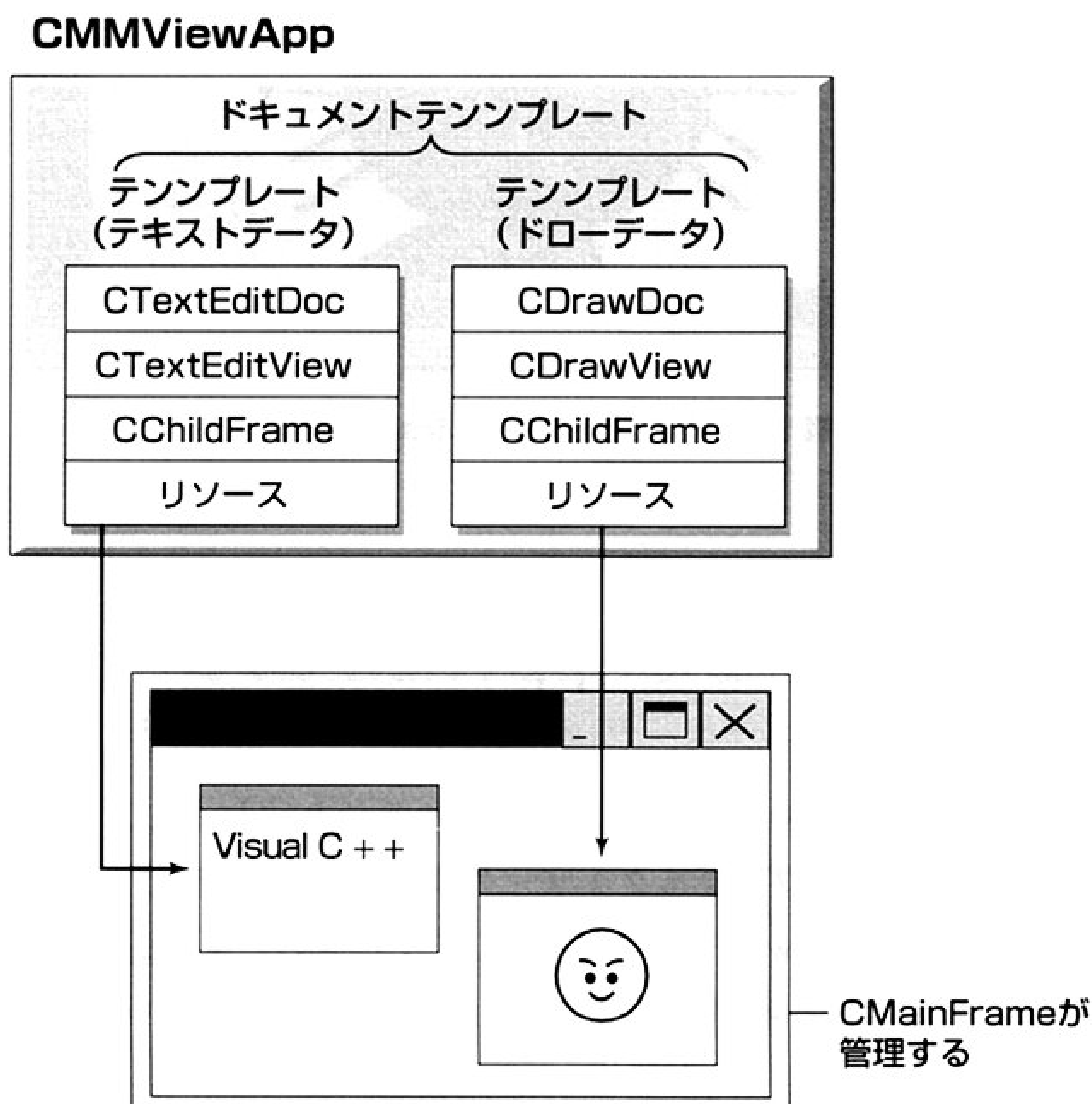


図1-1　主要クラスの関係図

それぞれのドキュメントタイプは独立して作ることができるので、まず本章でテキストエディタを作りながら、ドキュメント－ビュー・アーキテクチャを利用したアプリケーションの基本的な構成について説明します。また、ドキュメント－ビュー・アーキテクチャと並ぶMFCのもう1つの柱であるシリアライズについても説明します。2章では、ドローツールを作成しながら、複数のドキュメントタイプを扱う方法、ドキュメントのシリアライズの実際、MFCのポリシーに乗っ取ったクラスの設計について説明します。

それでは、テキストエディタ＋ドローツール＝マルチメディアビューアの完成図を横目に見つつ、作業を始めることにしましょう（**図1-2**）。



図1-2　完成したアプリケーションの画面

1.3 最初の一歩はAppWizardから

何はともあれ、プログラムのスケルトンを作成することにしましょう。メニューから［ファイル］－［新規作成］－［プロジェクトワークスペース］を選択し、プロジェクト名に「MMView」（Multimedia Viewerの略）と入力してください。<OK>ボタンをクリックしたらAppWizardが起動するので、それにしたがって設定項目をいくつか設定します。

●ステップ1

ここでは［作成するアプリケーションの種類］で［MDI］を選択します。種類の異なる複数のドキュメントを1つのウィンドウに表示するには、MDIを利用するのが簡単で

す。MMView では2種類のドキュメントタイプを扱いますから、MDI を利用します。ただし、AppWizard は1組のドキュメントクラスとビュークラスしか生成してくれないので、とりあえずテキストエディタのために AppWizard が生成したクラスを使うことにしましょう。もう1つのドキュメントタイプを扱うためのクラスは、あとで ClassWizard を使って作ります。

● **ステップ2**

ここではアプリケーションにデータベースへのアクセス機能を付加する設定を行うことができますが、MMView には必要ないので［データベースのサポート］に［しない］を選択します。なおデータベースのサポートを利用すると、ODBC や DAO を用いたデータベースへのアクセスを行う基本的なコードが AppWizard によってスケルトンに追加されます。MFC にはデータベースを利用するアプリケーションのために、CRecordView クラスや CRecordset クラスなどが用意されています。

● **ステップ3**

ここでは OLE 対応アプリケーションを作成するための設定を行うことができますが、やはり MMView には必要ないので、［どの複合ドキュメントをサポートしますか?］には［しない］を選択します。また［その他どのサポートをしますか?］についてもチェックをはずしておきます。なお OLE 対応アプリケーションは、複数のアプリケーションがあたかも1つのアプリケーションであるかのように連携して機能することができます。

● **ステップ4**

ここではその他もろもろの設定を行います。まず［アプリケーションへ組み込む機能］には、初期状態のまま、［ドッキングツールバー］、［初期ステータスバー］、［印刷および印刷プレビュー］、［3D コントロール］の4つにチェックを付けてください。
［MAPI（メッセージ API）］と［Windows ソケット］のチェックは2つともはずしておきます。これらはメールの送受信やインターネットへのアクセスを行うアプリケーションを作成するときに利用する機能です。
［最新ファイルの一覧に表示するファイルの数］は、［ファイル］メニューに最近オープンしたファイルの一覧を表示するときに、最大いくつまでファイルを並べるかを決めるものです。

● **ステップ5**

［ソースファイルのコメントを生成しますか?］は作成する EXE ファイルには直接影響しません。ただ、AppWizard や ClassWizard が生成するコードに自動的にコメントが追加されるだけです。2つの Wizard が生成してくれるコードは便利である半面理解せずに利用してしまいがちです。慣れないうちはコメントを付けておくとよいでしょう。

また、Professional Edition 以上の Visual C++ 6.0 を使用している場合は、ここでリンクする MFC ライブラリの種類を指定できます(Standard Edition の場合は、リンクできるのは共有 DLL だけなので、AppWizard にはこの表示はないことに注意)。[MFC ライブラリとのリンク]は[MFC の共有 DLL を使用]にチェックを付けてください。共有 DLL を使うと EXE ファイルのサイズを小さく抑えることができますが、別途 MFC42.DLL 等の DLL ファイルを用意しなければ動作しない EXE ファイルになります(通常「Windows¥System」フォルダに用意される)。Visual C++にはもちろん必要な DLL が添付されているので問題ありませんが、他のマシン上でも動かしたければ、DLL も同時に持っていかなければならないかもしれません。共有 DLL の代わりにスタティックライブラリを使うと、EXE ファイルのサイズは大きくなりますが、DLL を必要としません。この項目はスケルトンの生成後も変更が可能ですから、通常は共有 DLL を使用としておけばよいでしょう。

● **ステップ 6**

最後に AppWizard が生成するクラスとファイルの名前を設定します。AppWizard はプロジェクト名をもとにしてデフォルトのクラス名やファイル名を決めますが、MMView のように複数のドキュメントを扱うアプリケーションではそれぞれのクラスが果たす役割にそぐわない名前になってしまいます。ここでは**表 1-2** のようにクラス名を変更します。同時に、これらのクラスの実装ファイルのファイル名も変更しておいてください(**図 1-3**)。

クラス	古いクラス名	新しいクラス名	古いファイル名	新しいファイル名
ドキュメントクラス	CMMViewDoc	CTextEditDoc	MMViewDoc.cpp	TextEditDoc.cpp
ビュークラス	CMMViewView	CTextEditView	MMViewView.cpp	TextEditView.cpp

表 1-2 [クラス] ダイアログボックスでの変更部分

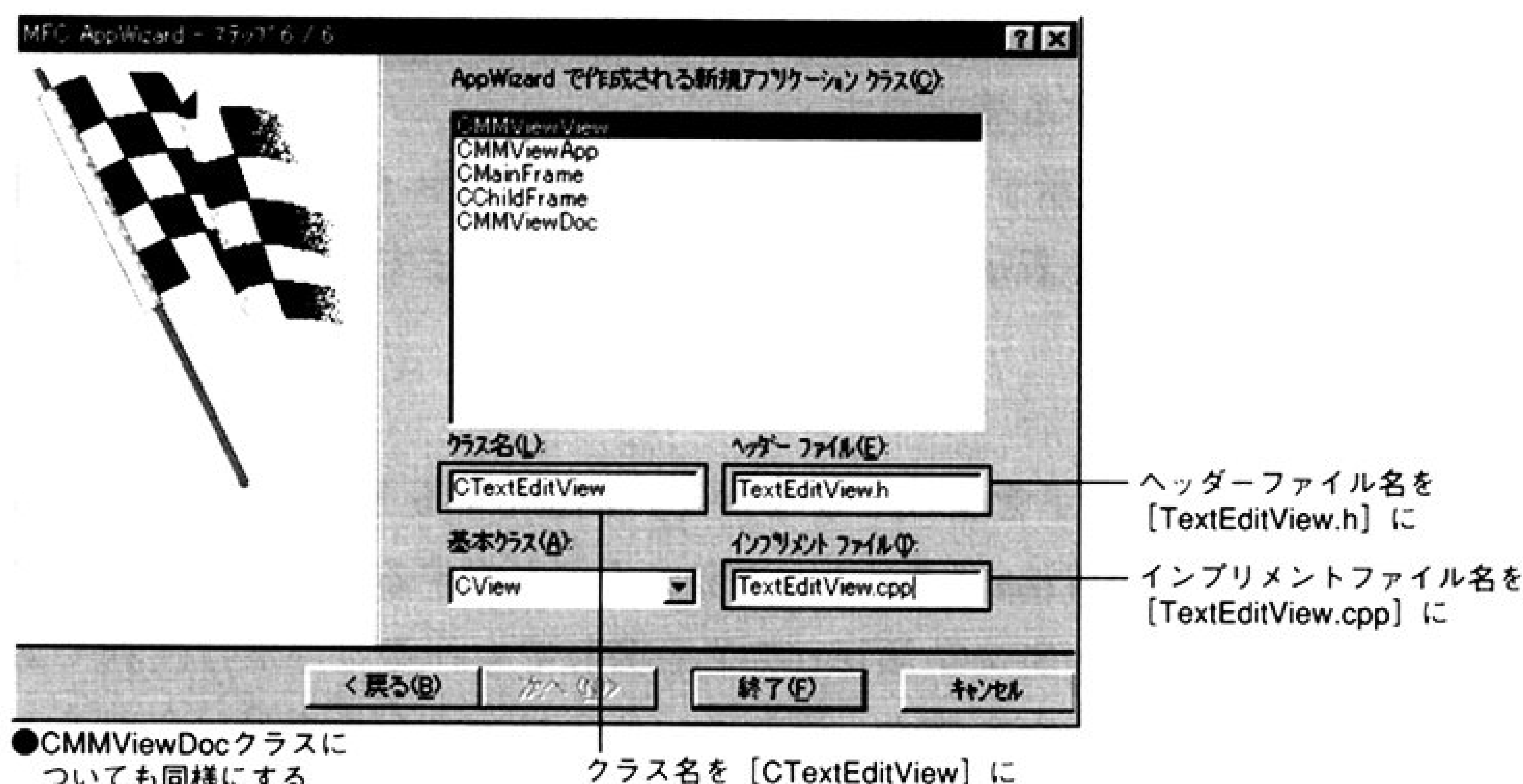


図 1-3 クラス名とファイル名の変更

以上の操作が終わったら、<終了>ボタン、続けて<OK>ボタンをクリックしてスケルトンを生成してください。スケルトンが作成されたら、次はドキュメントクラスとビュークラスの実装に取りかかることにしましょう。

1.4 たったの4行でテキストエディタのできあがり

テキストエディタにはどんな機能が必要でしょうか？ とにかく、ファイルの読み書き、テキストの挿入／削除ができることは絶対条件ですし、1画面に収まらないテキストならば、スクロールバーを使ってスクロールさせることも必要です。それから、クリップボードを使った処理や文字列の検索／置換などもできれば便利です。さらに、こうした処理を行うためには、いくつものダイアログボックスやメニューを用意する必要もあります。これだけの処理を行うためには膨大な量のコードが必要になることは想像に難くありません。なにしろ簡単にテキストの挿入といっても、キーボードからの入力／メモリ管理／文字列の表示など必要な処理はいくらでもあります。

少々不安になってきたかもしれませんが、もちろん上述した処理をすべて1から記述するような真似はしません。たとえば、エディットコントロールを使えば、テキストの編集はほとんどコントロール内で処理することができます。また、AppWizardが生成したコードをコンパイルしてみればわかるように、すでにメニューやファイルオープンダイアログなどは用意されています。しなければならないのはコントロールやフレームワークのサービスをうまく結び付けて、テキストエディタとして動作させることだけなのです。不安を取り除くために先に述べておきますが、AppWizardが生成したコードにたった4行のコードを付け加えるだけで、テキストエディタは完成です。あとはリソースを少々書き換えるだけです。しかも、MDIを利用しているので、マルチバッファ／マルチウィンドウをサポートした、マルチファイルテキストエディタがあっというまにできあがります。

気の早い読者のために、先に変更箇所を見せてしまいましょう（**リスト1-1**）。なぜたった4行だけの変更でマルチファイルテキストエディタができてしまうのか、変更箇所の少なさに反して、これを理解するためにはフレームワークのエッセンスともいうべき部分にまで触れなければなりません。それでは、この4行を理解するために、ドキュメントテンプレートとシリアライズについて説明していきます。

リスト 1-1 変更箇所のすべて

```
BOOL CMMView::InitInstance()
{
    ...略
    CMultiDocTemplate* pDocTemplate;
    pDocTemplate = new CMultiDocTemplate(
        IDR_MMVIEWTYPE,
        RUNTIME_CLASS(CTextEditDoc),
        RUNTIME_CLASS(CChildFrame), // カスタム MDI 子フレーム
        RUNTIME_CLASS(CEditView));  // CTextEditView を CEditView に変更
    AddDocTemplate(pDocTemplate);
    ...略
}

void CTextEditDoc::Serialize(CArchive& ar)
{
  POSITION pos = GetFirstViewPosition();  // 以下 3 行を追加
  CView* pEv = GetNextView(pos);
  ((CEditView*)pEv)->SerializeRaw(ar);
}
```

1.5 CEditViewクラスで楽しよう

まずは第 1 の変更点から見ていくことにしましょう。

さきほどわざわざ名前まで変更して CTextEditView クラスを作りましたが、実はこのクラスは使いません。というのは、まさにテキストエディタを作るためにあるかのようなクラスが MFC に用意されているからです。このクラスは「CEditView」といい、CCtrlView クラスの派生クラスです。CEditView クラスは非常に多機能なクラスで、内部でエディットコントロールを使うことにより、テキストの編集やクリップボードとのカット&ペースト、テキストの検索/置換などを一手に引き受けてくれます。CEditView クラスをビュークラスとして採用すると、たちどころにテキストエディタのできあがりです。もっとも、エディットコントロールには 64K バイト以上のテキストを編集できないという制限があるので、CEditView クラスを使ったテキストエディタで編集可能なファイルサイズは 64K バイトに制限されます。

それでは、どのようにすればビュークラスを CTextEditView クラスから CEditView クラスに変更できるのでしょうか? その答は MMView.cpp にある CMMViewApp::InitInstance 関数にあります。この中に次のような行があります。

```
CMultiDocTemplate* pDocTemplate;
pDocTemplate = new CMultiDocTemplate(
    IDR_MMVIEWTYPE,
    RUNTIME_CLASS(CTextEditDoc),
    RUNTIME_CLASS(CChildFrame), // カスタム MDI 子フレーム
    RUNTIME_CLASS(CTextEditView));  //CTextEditViewをCEditViewに変更
AddDocTemplate(pDocTemplate);
```

ここでアプリケーションが利用するビューやドキュメントのクラスを指定していることが直感的にわかるでしょう。この「CTextEditView」を「CEditView」に書き換えれば、ビュークラスとしてCEditViewクラスが使われるようになるのです。

もう少し細かく見てみましょう。まずnew演算子を使ってCMultiDocTemplateクラスのオブジェクトを作成しています。CMultiDocTemplateクラスのオブジェクトは**ドキュメントテンプレート**と呼ばれ、1つのドキュメントタイプを定義します。ドキュメントテンプレートは、以下のようにリソース／ドキュメントクラス／MDIの子ウィンドウのフレームウィンドウクラス／子ウィンドウのビュークラスの4つのパラメータから構成されていて、ドキュメントタイプの性格がここで決まります。作成したCMultiDocTemplateオブジェクトをAddDocTemplate関数を使用して、ドキュメントテンプレートのリストに追加しているわけです。ドキュメントテンプレートは実行時にウィンドウを作成するときにアプリケーションクラスのオブジェクトから参照されるようになっています。

```
new CMultiDocTemplate(  //MDIドキュメントテンプレートの作成
    リソース,            //CMultiDocTemplateクラスのコンストラクタへの引数
    ドキュメントクラス,
    子ウィンドウ用のフレームウィンドウクラス,
    子ウィンドウ用のビュークラス
);
```

リソースについてはあとで詳しく説明することにして、残りの3つのクラスの説明をしてしまいましょう。登録するクラスは、RUNTIME_CLASSマクロを使って指定します(RUNTIME_CLASSマクロについては次節で詳しく説明する)。3つのクラスは前から、ドキュメントクラス／子ウィンドウのフレームウィンドウクラス／子ウィンドウのビュークラスの順に指定します。ビュークラスとドキュメントクラスには、直接CViewクラスやCDocumentクラスを指定せずに、その派生クラスを指定しなければなりません。また、フレームウィンドウクラスには、デフォルトではCMDIChildWndクラスの派生クラスであるCChildFrameクラスが使用されます。CChildFrameクラスはAppWizardが定義したクラスですが、その中身は空なのでCMDIChildWndクラスをそのまま利用した場合と変わりありません。

作成したドキュメントテンプレートは、CWinApp::AddDocTemplate関数を使ってドキュメントテンプレートのリストに登録しておきます。すると、あとで［ファイル］－［新規作成］が選択されたときなど、新規にドキュメントを作成する際にフレームワークから参照され、その結果ウィンドウが新規に作成／表示されるようになります。

ドキュメントテンプレートに登録するビュークラスを変更することで、CEditViewクラスが利用できるようになることがわかったでしょうか？ これが第1の変更点の理由です。

Q1 なぜAddDocTemplate関数はInitInstance関数の中で呼ばれているのですか？
A1 ドキュメントテンプレートはインスタンスの初期化時に一度だけリストに登録すればいいからです。

何らかの処理を行いたいというときに、その処理を記述する関数をMFCが提供する多くの（仮想）関数の中から選択する基準は、行う処理の内容と関数が呼ばれるタイミングにあります。たとえば、ここで行う処理はドキュメントテンプレートというオブジェクトを作成しそれをテンプレートのリストに登録することです。ドキュメントテンプレートは最初にドキュメントを開く前に最低でも1つは登録されていなければいけませんし、一度作成されたドキュメントテンプレートはアプリケーションが終了するまで削除することもできません。また、同じドキュメントテンプレートを二度登録しても無意味です。そこで、アプリケーションの起動時に一度だけ呼び出されるInitInstance関数でドキュメントテンプレートの定義を行うのです。

Q2 ドキュメントテンプレートオブジェクトをnew演算子で作成しているにもかかわらず、なぜどこでもdeleteしていないのですか？
A2 フレームワークが自動的に削除してくれるからです。

一般にnew演算子を使ってオブジェクトを作成した場合は、delete演算子を使ってオブジェクトを削除する必要があります。

しかし、ドキュメントテンプレートのオブジェクトについては、new演算子で作成しているにもかかわらず、どこにもこれを削除するコードは見当たりません。これは別にオブジェクトを削除する必要がないわけではなく、フレームワークの中で後始末が付けられるようになっているからです。これは比較的特殊な例なので、普段は忘れずにdelete演算子でオブジェクトを削除するようにしてください。

なお、逆にフレームワークの中で作成されたオブジェクトを勝手に削除してしまわないように気をつけましょう。

1.6 RUNTIME_CLASS

少々話が脇道にそれてしまいますが、ここでさきほど出てきたRUNTIME_CLASSマクロについて詳しく説明することにします。もう一度リストを見てみましょう。

```
CMultiDocTemplate* pDocTemplate;
pDocTemplate = new CMultiDocTemplate(
    IDR_MMVIEWTYPE,
    RUNTIME_CLASS(CTextEditDoc),
    RUNTIME_CLASS(CChildFrame), // カスタム MDI 子フレーム
    RUNTIME_CLASS(CTextEditView));  // CTextEditView を CEditView に変更
AddDocTemplate(pDocTemplate);
```

RUNTIME_CLASSマクロはCMultiDocTemplateクラスのコンストラクタの引数に使用されていますが、実はこのマクロを利用することによりコンストラクタにはクラスのオブジェクトではなく、クラス自体が渡されているのです。いうなれば、変数ではなく、変数の型が渡されているようなものです。C++言語では、**RTTI**（**RunTime Type Information**：**ランタイム型情報**）により、実行時の型を扱うことができるようになりましたが、MFCでは実行時に型を扱うためにこの機能を使用してはいません。かといって別にVisual C++の処理系が拡張されているわけでもなく、いくつかのマクロを組み合わせて、擬似的にクラスというオブジェクトの型を関数の引数として渡すことができるようになっているのです。

実行時に型情報を扱えるようなしくみを用意した大きな原因としてあげられるのは、Windowsアプリケーション（とりわけ、MDIアプリケーション）では、ユーザーが[ファイル]－[新規作成]などを選択したときに動的にウィンドウを作成するということです。MFCを利用した場合、1つのウィンドウはフレーム／ビュー／ドキュメント／リソースから構成されることはすでに述べましたが、アプリケーションはこれらの情報をドキュメントテンプレートの形で管理し、必要なときに必要なだけそのオブジェクト（すなわちウィンドウ）を作成するのです。とくに複数のドキュメントタイプを扱ったり、1つのドキュメントに複数のビューがあるようなアプリケーションの場合には、ユーザーの選択によってどのドキュメントテンプレートを利用してウィンドウを作成するかはアプリケーションの実行時にならないとわかりません。このように、実行時に初めて作成するオブジェクトの型（クラス）が決定するような場合には、オブジェクトの型を関数の引数として渡せることが必要になってきます。

MFCでは、これをRUNTIME_CLASSマクロを使って実現しています。RUNTIME_CLASS（<クラス名>）マクロは、CRuntimeClassクラスのオブジェクトへのポインタを返します。<クラス名>に対応したCRuntimeClassクラスのオブジェクトを使うと、**表1-3**にあげるマクロの利用方法によって、3種類のサービスのいくつかを受けることがで

種類	クラスの定義部	クラスの実装部	サービス
DYNAMIC	DECLARE_DYNAMIC	IMPLEMENT_DYNAMIC	isKindOf 関数の利用
DYNCREATE	DECLARE_DYNCREATE	IMPLEMENT_DYNCREATE	オブジェクトの動的な作成
SERIAL	DECLARE_SERIAL	IMPLEMENT_SERIAL	シリアライズ

表 1-3 3 種類のサービスを提供するマクロ

きます。先に述べた動的なオブジェクトの作成は、3 種類のサービスのうちの 1 つです。

表 1-3 に示したマクロは、2 つで 1 組になっていて、DECLARE_~マクロはクラスの定義部、IMPLEMENT_~マクロはクラスの実装部で使います。たとえば、CTextEditDoc クラスでは**リスト 1-2** と**リスト 1-3** のようにして DECLARE_DYNCREATE マクロと IMPLEMENT_DYNCREATE マクロを利用しています。

リスト 1-2 クラス定義内で利用するマクロ (TextEditDoc.h)

```
class CTextEditDoc : public CDocument {
private:  // シリアライズ機能からのみ作成します。
  CTextEditDoc();
  DECLARE_DYNCREATE(CTextEditDoc) // クラス定義部でのマクロの利用

// アトリビュート
public:
  ...
}
```

リスト 1-3 クラスの実装内で利用するマクロ (TextEditDoc.cpp)

```
...
////////////////////////////////////////
// CTexteditDoc

IMPLEMENT_DYNCREATE(CTextEditDoc, CDocument)  // 実装部でのマクロの利用

BEGIN_MESSAGE_MAP(CTextEditDoc, CDocument)
...
```

クラスの定義に、いずれのマクロも利用していない場合には、そのクラスを RUNTIME_CLASS マクロの引数として与えることはできません。

DECLARE_DYNAMIC マクロと IMPLEMENT_DYNAMIC マクロを利用したクラスでは、IsKindOf 関数を使って、あるオブジェクトがどのクラスから作られたものなのか調べることができます。これを利用すると、あるクラスのオブジェクトへのポインタが、本

当にそのクラスのオブジェクトを指しているのか、それともその派生クラスのオブジェクトを指しているのかを知ることができます。たとえば、以下のような場合です。ここでは、CDocument クラスのオブジェクトへのポインタが渡された関数の中で、そのオブジェクトの本当の型（クラス）を判断するのに IsKindOf 関数を利用しています。

```
void Foo(CDocument* pDoc){
  if (pDoc->IsKindOf(RUNTIME_CLASS(CTextEditDoc)))
  {
    <pDoc が CTextEditDoc クラスのオブジェクトを指している場合の処理>
  }
  else
  {
    <pDoc が CTextEditDoc クラス以外のオブジェクトを指している場合の処理>
  }
}
```

DECLARE_DYNCREATE マクロと IMPLEMENT_DYNCREATE マクロを利用したクラスは、動的にオブジェクトを作成できるようになります。CView クラスや CDocument クラスの派生クラスは、フレームワークによって動的にオブジェクトが作成されるので、かならずこのマクロを利用しなければいけません。

DECLARE_SERIAL マクロと IMPLEMENT_SERIAL マクロを利用したクラスは、＞＞演算子や＜＜演算子を使ってシリアライズできるようになります（シリアライズについては次節以降で説明する）。なお、～_DYNCREATE マクロは～_DYNAMIC マクロの機能をすべて含んでおり、～_SERIAL マクロは～_DYNCREATE マクロの機能をすべて含んでいます。

1.7 ドキュメントクラスの実装

ビューの実装はドキュメントテンプレートの構成を変更して、ビュークラスに CEditView クラスを利用するだけでした。次にビューと対になるドキュメントの実装について説明しましょう。この２つの実装が済んで初めてプログラムは意味のあるものとなるのです。さて、ドキュメントの実装は、一般にメンバ変数の追加と、Serialize 関数の記述がその中心的な作業となります。メンバ変数は、ドキュメントの本体（たとえばテキストファイル）やその属性（テキストファイルのサイズなど）を保存するのに使います。Serialize 関数は、MFC が提供する仮想関数の１つで、ドキュメントの内容をアプリケーションとファイルとの間でやり取りするのに使います。ドキュメントクラスの実装を始める前に、まず Serialize 関数について少し説明しましょう。

● Serialize関数

MFCが取り入れている重要な概念の1つが**シリアライズ**です。シリアライズの目的は、「保存し、再利用する価値のあるすべてのクラスには、統一されたファイル入出力のインターフェイスを提供する」というところにあります。そして、そのインターフェイスにあたるのがSerialize関数です。

たとえば、ドキュメントクラスには、保存し、再利用する価値のあるデータが含まれていることはいうまでもありません。そこでフレームワークはドキュメントクラスのオブジェクトをファイルに保存したり、ファイルから読み込むときに、ドキュメントクラスのSerialize関数を呼び出します。そして、プログラマがフレームワークの規定する仕様に従ってSerialize関数を実装すれば、ファイルとの入出力が簡単に行えるようになります。

それでは、Serialize関数を使って決められているシリアライズの統一的なインターフェイスとはどのようなものなのでしょう。ここでは、AppWizardが生成したSerialize関数を例に説明します。AppWizardが生成したSerialize関数は、**リスト1-4**のようなコードです。なぜこのようなコードがすでに関数の本体に用意されているのでしょう。

リスト1-4　変更前のCTextEditDoc::Serialize関数

```
void CTextEditDoc::Serialize(CArchive& ar)
{
  if (ar.IsStoring())
  {
    // TODO:この位置に保存用のコードを追加してください。
  }
  else
  {
    // TODO: この位置に読み込み用のコードを追加してください。
  }
}
```

実は、ドキュメントクラスのSerialize関数はプログラマが直接呼び出すのではなく、フレームワークによって呼び出されるのです。よって、Serialize関数を実装する際には、その引数と返り値をフレームワークの期待どおりに扱うことが重要になってきます。ここでは返り値はvoidですから考える必要はありません。問題は引数です。引数にはCArchiveクラスのオブジェクトarがフレームワークから渡されています。CArchiveクラスは、ファイル入出力を行うためにMFCが提供しているクラスで、シリアライズと密接に関連しています。そこで、次項では少々寄り道をしてCArchiveクラスについて解説します。

● CArchiveクラス

C言語によるプログラミングの経験がある方は、標準入出力を使ったファイルアクセスについてはよくご存じでしょう。これはOSやシェルによって実現されていたパイプやリダイレクトといった機能を前提としたインターフェイスでした。従来のコマンドラインインターフェイスを用いた場合には、こうしたインターフェイスは非常に便利で使いやすいものでしたが、WindowsのようなGUIベースのシステムでは有効に機能しないであろうことは容易に想像できます。またVisual C++のようにC++言語を使う場合にも、C言語とは違った、C++言語に適したインターフェイスが求められます。

こうした要求に応えるのがCArchiveクラスです。CArchiveオブジェクトを作成すると、そのオブジェクトは入出力の対象となるファイルと1対1で対応付けされ、次のようなインターフェイスで、ファイルに対して入出力を行うことができるようになります。そして、この入出力の中心となるのが、>>演算子(読み込み)と<<演算子(書き出し)です。

```
int number = 5;
CString message = "Select number:";
if (ar.IsStoring())
  ar << message << number; // "Select number", 5 を出力
else
  ar >> message >> number; // CString 型, int 型を入力
```

>>演算子と<<演算子といえば、C言語ではint型の値を引数に取るビットシフト演算子でした。しかしC++言語ではそれだけではありません。C++言語では、関数と同様、演算子の動作もプログラマが自由に定義でき、さらに、同じ演算子でも引数の型によって異なる動作を定義できるようになっています。2つのint型変数を引数として>>演算子を使えば、C言語と同様にビットシフト演算が行われます。しかし上の例のように、左項をCArchiveクラスのオブジェクト、右項をint型変数として>>演算子を使えば、CArchiveクラスのメンバ関数として定義されている>>演算子が使われることになり、ファイルとの入出力が行われます。

```
int a, b;
CArchive ar;
a << b;   // ビットシフト演算
ar << a;  // ファイルへの出力
ar >> a;  // ファイルからの入力
```

このようなCArchiveクラスのインターフェイスには2つのメリットがあります。1つはオブジェクトの型チェックがコンパイラによって自動的に行われることです。たとえば、printf関数やscanf関数を使ったファイルの入出力では、入出力フォーマット(%sや%cなど)と引数の整合性はプログラマが管理する必要がありましたが、CArchiveクラスを使えば、入出力を行うオブジェクトの型に対応した関数を呼び出すようにコンパイラが処理してくれます。もう1つは、プログラマが作ったクラスのオブジェクトに対するインター

フェイスを容易に追加できることです。printf関数やscanf関数では、int型やchar型といったプリミティブな型しか入出力フォーマットを指定できませんでしたが、CArchiveクラスを使えば、プログラマが記述したクラスの入出力をint型やchar型と同じように>>演算子や<<演算子を使って行うことができます。

Serialize関数の標準的な構造

Serialize関数の引数に渡されたCArchiveオブジェクトarを使えば、ファイルの入出力ができることはわかりました。しかし、Serialize関数の中で、どのファイルにアクセスするのかはどうやって知るのでしょうか? なにせ、Serialize関数の引数にはファイル名など渡されていないのです。

実はそんなものはプログラマが管理する必要はないのです。というのは、Serialize関数が呼び出されるときには、フレームワークによってarに対応するファイルはすでにオープンされているからです。ここに至るまでにフレームワークは、［ファイルを開く］などのダイアログをオープンし、そこでユーザーが指定したファイルをオープンしてくれるのです。つまり、Serialize関数の中では、プログラマはファイルの入出力に関するコードだけを記述すればよいのです。Serialize関数が終了すれば、ファイルのクローズもフレームワークが行ってくれます。

ダイアログボックスを使ったファイルの選択やファイルのオープンといった面倒な作業をフレームワークが肩代わりしてくれるのは嬉しい限りですが、そうすると次の問題が発生します。C言語でfopen関数などを使ってファイルをオープンするときには、ファイル名ともう1つ、読み込みや書き込みといったモードの指定を同時に行いました。しかしMFCを使ったプログラムでは読み込みか書き込みかにかかわらず、呼び出される関数は常にSerialize関数です。それでは、Serialize関数の中では、どのように読み込みと書き込みの処理を区別して行えばよいのでしょうか?

この区別をするコードこそがAppWizardが生成したSerialize関数にあったコードなのです。というわけで、ここでもう一度AppWizardによるコードを見てみることにしましょう（**リスト1-5**）。

リスト1-5 AppWizardが生成したSerialize関数

```
void CTextEditDoc::Serialize(CArchive& ar)
{
  if (ar.IsStoring())
  {
    // TODO:この位置に保存用のコードを追加してください。
  }
  else
  {
```

```
    // TODO: この位置に読み込み用のコードを追加してください。
  }
}
```

CArchiveクラスには、CArchiveクラスのオブジェクトが読み込みと書き込みのどちらのモードでオープンされているかを調べる、IsStoring関数とIsLoading関数という2つのメンバ関数が用意されています。IsStoring関数は対応するファイルが書き出し用にオープンされていればTRUEを、読み込み用にオープンされていればFALSEを返します。またIsLoading関数はIsStoring関数とは逆に、読み込み用にオープンされていればTRUEを、書き出し用にオープンされていればFALSEを返します。すべてのSerialize関数では、かならずこのどちらかの関数を使って処理を振り分ける必要があります。

● SerializeRaw関数

前項では一般的なSerialize関数の構造について説明しましたが、実はCTextEditDoc::Serialize関数に次のコードを記述するだけで、本節での目的の作業は完了してしまいます（**リスト1-6**）。今までに説明したようなシリアライズが実際にはどのようにして行われているのかについては2章で見てもらうことにして、ここではMFCが提供する機能を使って楽をしてしまいましょう。

リスト1-6　CTextEditDoc::Serialize関数（TextEditDoc.cpp）

```
CTextEditDoc::Serialize(CArchive& ar)
{
  POSITION pos = GetFirstViewPosition();   // ビューオブジェクトを取得する準備
  CView* pEv = GetNextView(pos);           // ビューオブジェクトを取得
  ((CEditView*)pEv)->SerializeRaw(ar);     // SerializeRaw関数を呼び出す
}
```

ポイントはCEditView::SerializeRaw関数にあります。この関数はCEditViewクラスのメンバ関数として呼び出されていることからもわかるように、CEditViewクラスのサービスとして用意されています。つまりCEditViewクラスは、ビュークラス本来の目的であるユーザーインターフェイスの管理以外にも、シリアライズを含めたドキュメントの管理まで受け持っているのです（CEditView::SerializeRaw関数内ではCArchive::IsStoring関数を利用して一般的なSerialize関数と同様な処理を行っている）。このため、CTextEditDocクラスにはメンバ変数を追加する必要もなければ、ファイルとの入出力を行うコードを記述する必要もありません。

とはいえ、それほど簡単にSerializeRaw関数を呼び出せるわけではありません。なぜなら、SerializeRaw関数はCEditViewクラスのメンバ関数であり、CTextEditDocクラスのメンバ関数であるSerialize関数からは直接呼び出すことはできないからです。そこで、まずはCEditViewクラスのオブジェクトへのポインタを手に入れる必要があります。そこで登場するのが、本章の冒頭で**表1-1**にあげた**GetFirstViewPosition関数**と**GetNextView関数**です。これら2つの関数は、ドキュメントクラスからドキュメントテンプレートを介して関連付けされたビュークラスのオブジェクトを知るために使います。逆にビュークラスで、ビューに関連するドキュメントクラスのオブジェクトを知るためには、各ビュークラスでオーバーライドされたGetDocument関数を使います。

Q1 GetNextView(GetFirstViewPosition())と記述すれば1行で済むのでは？
A1 実はそうはいきません。

なぜならGetNextView関数の引数の型はPOSITION&というPOSITION型の参照*1だからです。C++言語には、参照の変数に代入するときの右辺値は変数でなければならないのですが、関数の引数が参照のときにもこれを守る必要があります。つまり、参照の引数を取る関数には、変数を介さなければ値を渡せないのです。

2行に分けて書かなければいけないのは面倒ですが、もちろん理由もなく、このようになっているわけではありません。GetNextView関数の内部では、引数に渡された変数(pos)の値を参照したあとに、次のビューオブジェクトを指すようにその値を変更しているのです。値を変更するからには、関数の返り値ではなく変数を渡さなければならないのは当然です。こうした仕様にすることで、以下のように繰り返しGetNextView関数を呼び出すだけで順にビューオブジェクトへのポインタを手に入れることができるのです。

```
CView* pv;
POSITION pos = GetFirstViewPosition();
while (pos) {              // これ以上ビューがなければ pos==NULL
  pv = GetNextView(pos);   // ビューを入手。pos は次のビューを指す
  <ビューに対する操作>
}
```

ただ、このプログラムでは1つしかビューを利用していないために、こうしたインターフェイスの利点よりも二度手間の面倒さが目立ってしまっているのです。

*1 参照についてはAppendix Aを参照。

Q2 CEditView クラスがドキュメントの管理まで行っているが、これは本来ドキュメントクラスで行うべき作業ではないか？
A2 まったくそのとおりです。

CEditView クラスは非常に特殊な構成をしたクラスで、1つのクラスでビュークラスとドキュメントクラスの両方の機能を持っています。これは、CEditView クラスが継承されることをとくに考慮していなかったり、複数のビューに対応したドキュメント管理をするつもりがないためです。つまり、CEditView クラスはすでにそれだけで完成されたクラスであるために、このようになっているのです。こうしたクラスを設計することはできますが、後々のことを考えて、できるだけビューとドキュメントは分けて設計した方がよいでしょう。

自分でクラスを設計するときは、どこまでをビュークラスで、どこまでをドキュメントクラスで扱ったらよいかは悩むところではあると思いますが、2つ以上のビューを想定してみるとはっきりしてきます。ビューに依存せずに常に必要になる情報はドキュメントクラスで管理し、それ以外の情報はビュークラスで管理すればよいのです。

以上で変更を加える4行のコードについての説明は終わりです。あとはリソースに手を加えるだけでテキストエディタができあがります。

1.8 リソースの編集

少し話がもとに戻りますが、AddDocTemplate 関数の説明で、ドキュメントテンプレートを構成するパラメータの1つにリソースがあると述べました。実はこのパラメータは、メニューやアイコン、ストリングテーブルなど、いくつものリソースを同時に指定しているのです。これらのリソースをこれからリソースエディタで編集していくことにします（**図1-4**）。

● AppWizard の作ったリソース

では、エディタ用のリソースの作成を行うことにしましょう。［表示］－［プロジェクトワークスペース］を選択してワークスペースウィンドウを表示させ、［ResourceView］タブをクリックして、リソース一覧を表示してください。今表示されているリソースはAppWizard が生成した、いわばデフォルトのリソースです。これをアプリケーションに合わせてリソースエディタを使って書き換えるのがこれからの作業です。

ところで、これらのリソースすべてがドキュメントテンプレートに登録されて利用されるわけではなく、あるものはフレームワークから参照され、あるものはプログラマが記述し

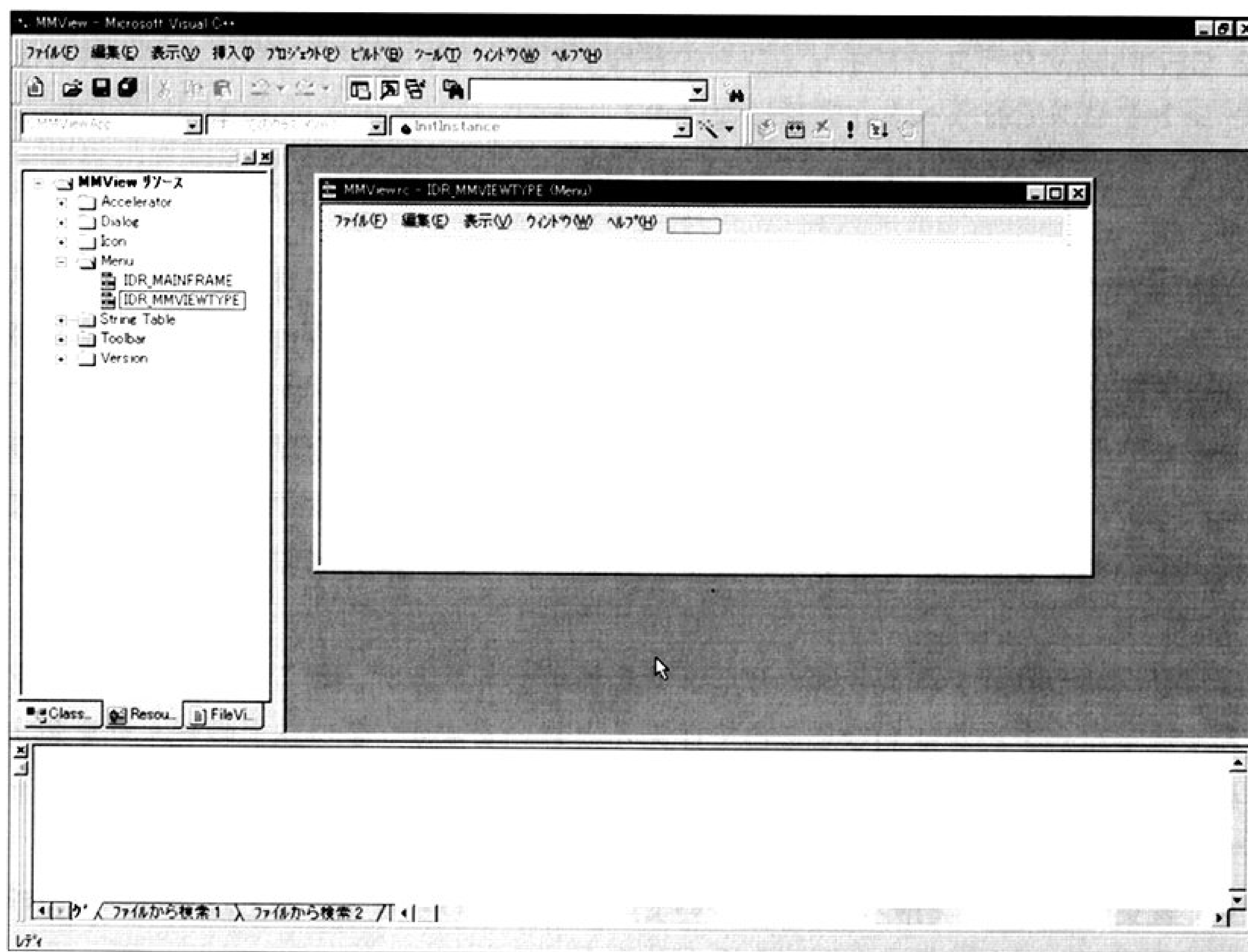


図 1-4 リソースエディタ

たコードから参照されます。では、ドキュメントテンプレートに登録されるリソースとは、どのことなのでしょう？ もう一度 CMMViewApp::InitInstance 関数で AddDocTemplate 関数を呼び出している部分を見てください。

```
CMultiDocTemplate* pDocTemplate;
pDocTemplate = new CMultiDocTemplate(
    IDR_MMVIEWTYPE,
    RUNTIME_CLASS(CTextEditDoc),
    RUNTIME_CLASS(CChildFrame), // カスタム MDI 子フレーム
    RUNTIME_CLASS(CTextEditView)); // CTextEditView を CEditView に変更
AddDocTemplate(pDocTemplate);
```

CMultiDocTemplate クラスのコンストラクタに渡す最初の引数がリソースの指定ですから、IDR_MMVIEWTYPE が登録するリソースを示しています。そこで、シンボルブラウザを使って「IDR_MMVIEWTYPE」というリソース ID を探してみましょう。メニューから［表示］－［シンボルブラウザ］を選択するとシンボルブラウザが起動されます。シンボルブラウザには現在定義されているシンボルの一覧が表示されるので、この中から IDR_MMVIEWTYPE を選びます。するとアイコン／メニュー／ストリングテーブルの 3 つのリソースに同じ IDR_MMVIEWTYPE というリソース ID が割り当てられていることがわかります（**図 1-5**）。

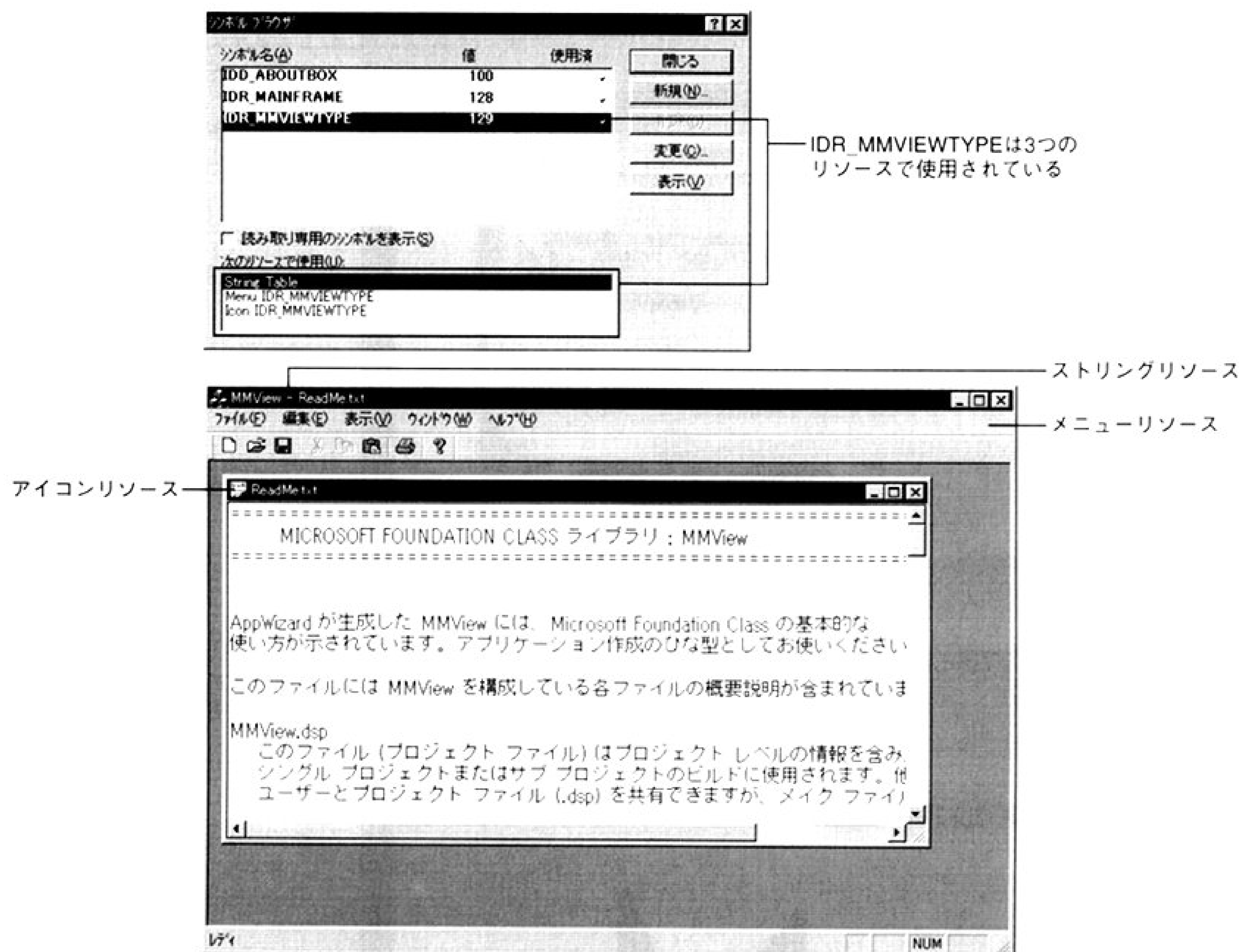


図 1-5　テンプレートに登録するリソースのタイプ

これら3つのリソースともう1つ、アクセラレータテーブルを合わせた4種類のリソースがドキュメントテンプレートに登録されるリソースです（**表 1-4**）。これらのリソースがドキュメントテンプレートに登録されるということは、ドキュメントテンプレートごとに異なるリソースが使われるということです。たとえば、MDI アプリケーションでは、アクティブな子ウィンドウが変更されると、メインフレームウィンドウに表示されるメニューもそれに合わせて変更されるのです（**図 1-6**）。

リソースの種類	用途
アイコン	子ウィンドウを最小化したときに表示されるアイコン
メニュー	子ウィンドウをアクティブにしたときに使われるメニュー
ストリングテーブル	ウィンドウタイトルやファイルタイプに表示される文字列
アクセラレータテーブル	ショートカットキー

表 1-4　ドキュメントテンプレートに登録するリソース

メニューだけでなく、アイコンやアクセラレータテーブルも、ドキュメントテンプレートに登録されたものが使われます。

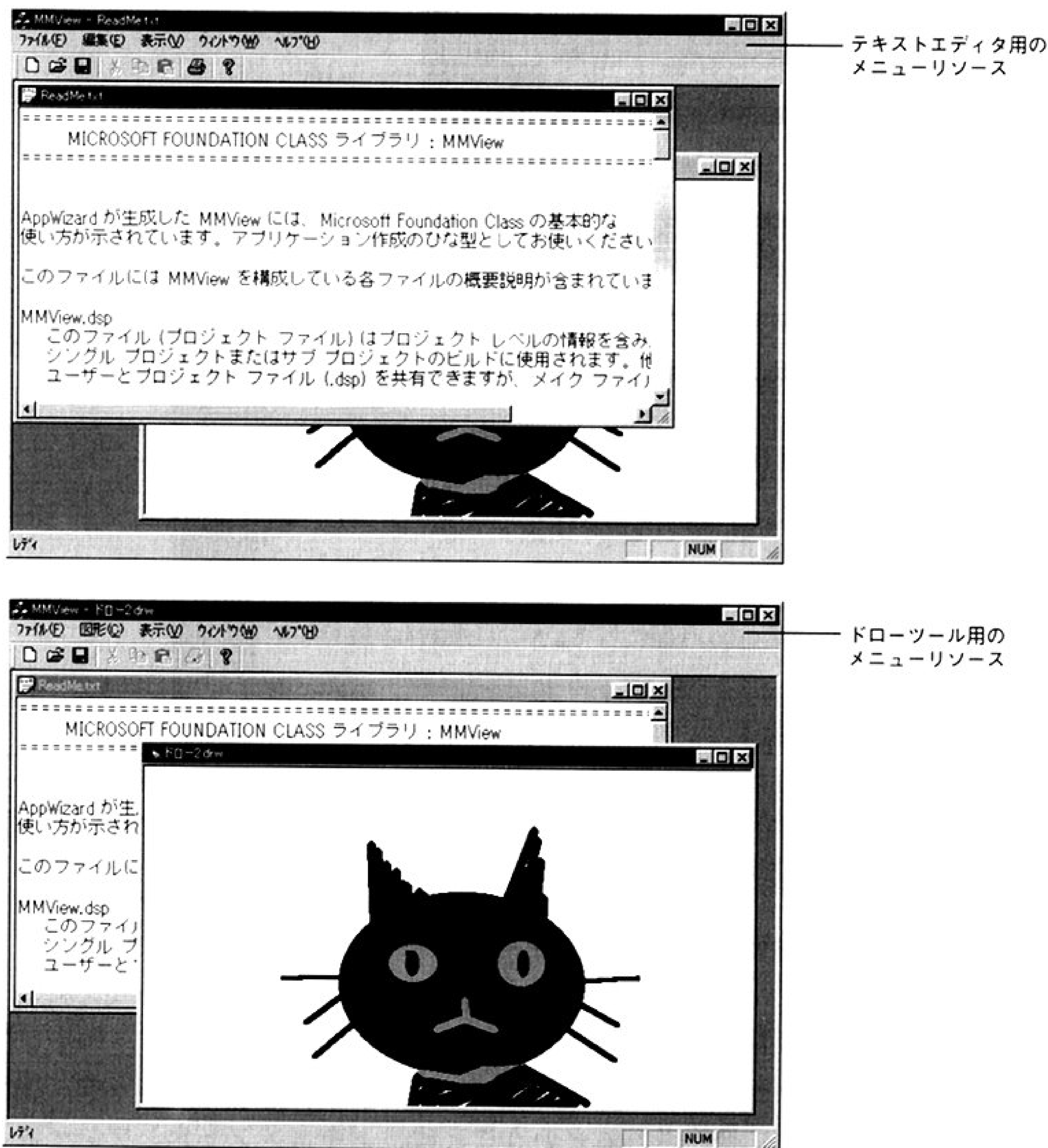


図 1-6　ドキュメントタイプごとに異なるリソース

では、子ウィンドウが1つも表示されていない状態では、どのリソースが使われるのでしょう？ もう一度**図 1-5** を眺めてみると、「IDR_MAINFRAME」というリソース ID があります。名前からもわかるように、このリソース ID はアプリケーションのメインフレームウィンドウが利用しています。実は、子ウィンドウが1つも表示されていないときは、この ID で指定されるリソースが使われます。IDR_MAINFRAME はドキュメントテンプレートに登録されてから参照されるのではなく、メインフレームウィンドウを作成するために LoadFrame 関数を呼び出すときにその引数として以下のように参照されています。

```
CMainFrame* pMainFrame = new CMainFrame;
if (!pMainFrame->LoadFrame(IDR_MAINFRAME))
  return FALSE;
```

リソースの種類	用途
アクセラレータ	ドキュメントに依存しないショートカットキー
ビットマップ	ツールバーに表示されるビットマップ
アイコン	アプリケーションを最少化したときに表示されるアイコン
メニュー	子ウィンドウが存在しないときに使われるメニュー
ストリングテーブル	メインフレームウィンドウのタイトル

表 1-5 メインフレームウィンドウに付随するリソース

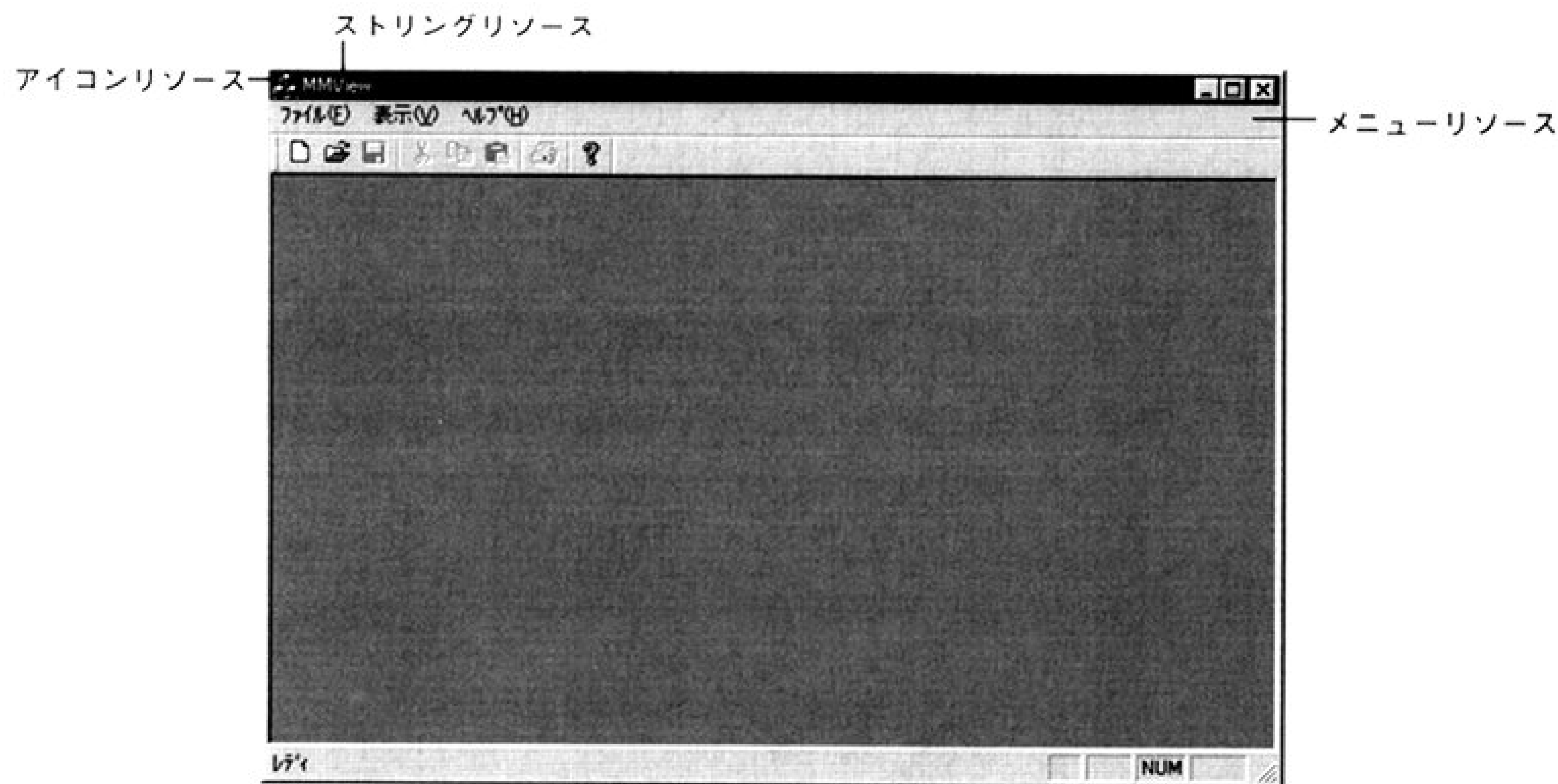
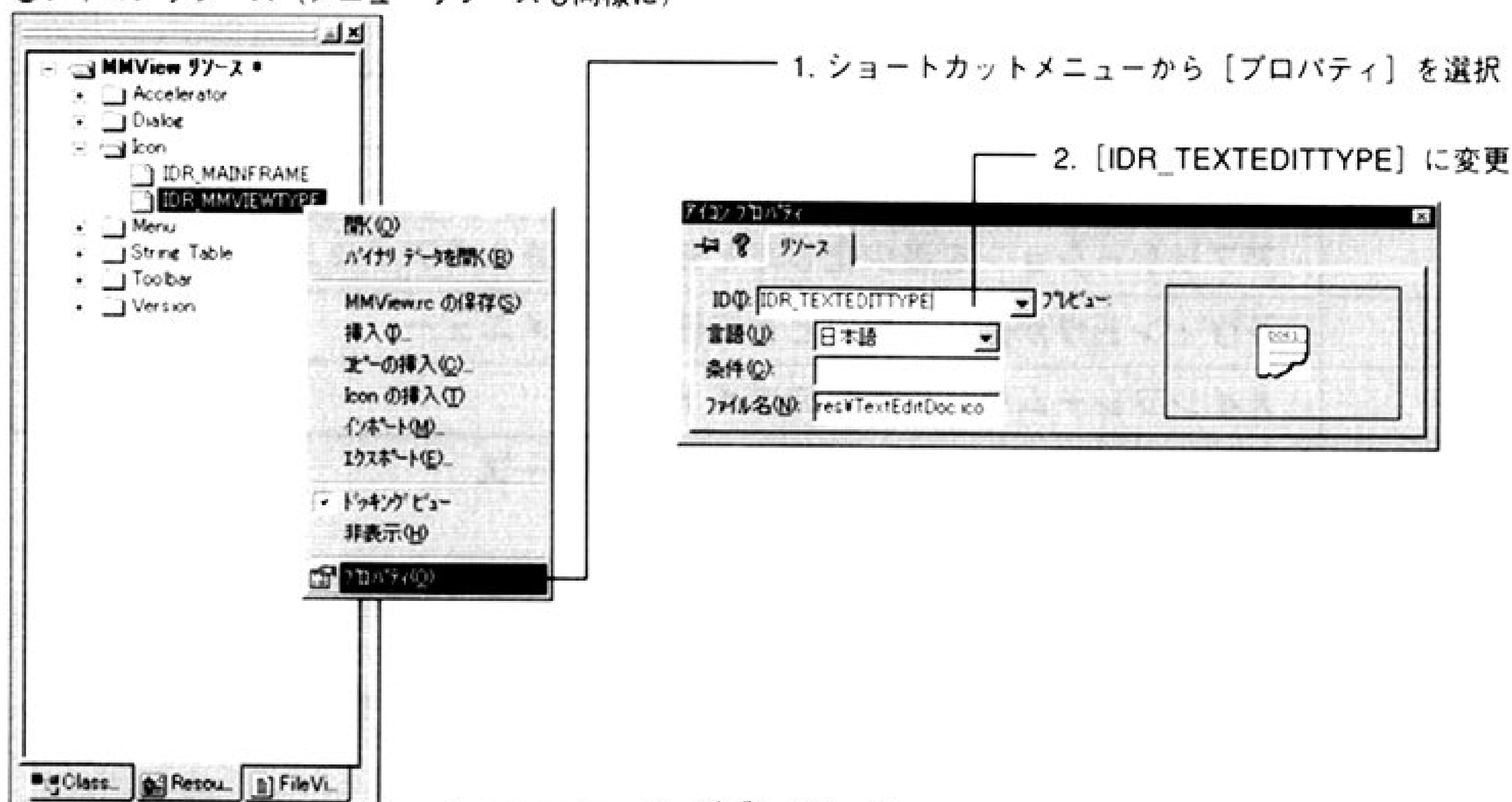


図 1-7 メインフレームウィンドウに付随するリソース

AppWizard が生成したリソースの説明が済んだところで、IDR_MMVIEWTYPE が示すリソースを編集することにします。しかし、その前に、IDR_MMVIEWTYPE という名前はそぐわないので名前を変更することにしましょう。というのは、MMView というのはアプリケーションの名前であるにもかかわらず、IDR_MMVIEWTYPE が表しているのは 1 つのドキュメントテンプレートに関連付けられたリソースです。複数のドキュメントテンプレートを使用する場合に、1 つのドキュメントテンプレートにアプリケーション名を付けるのは適当とはいえないでしょう。そこで、IDR_MMVIEWTYPE という名前のリソースは、ワークスペースウィンドウからその名前をすべて「IDR_TEXTEDITTYPE」に変更してしまいましょう（**図 1-8**）。

また、これに合わせてプログラム上でも IDR_MMVIEWTYPE を参照していた箇所を IDR_TEXTEDITTYPE に書き換えます。といっても、実際に変更する必要のある部分は**リスト 1-7** に示したドキュメントテンプレートを登録している部分だけです。



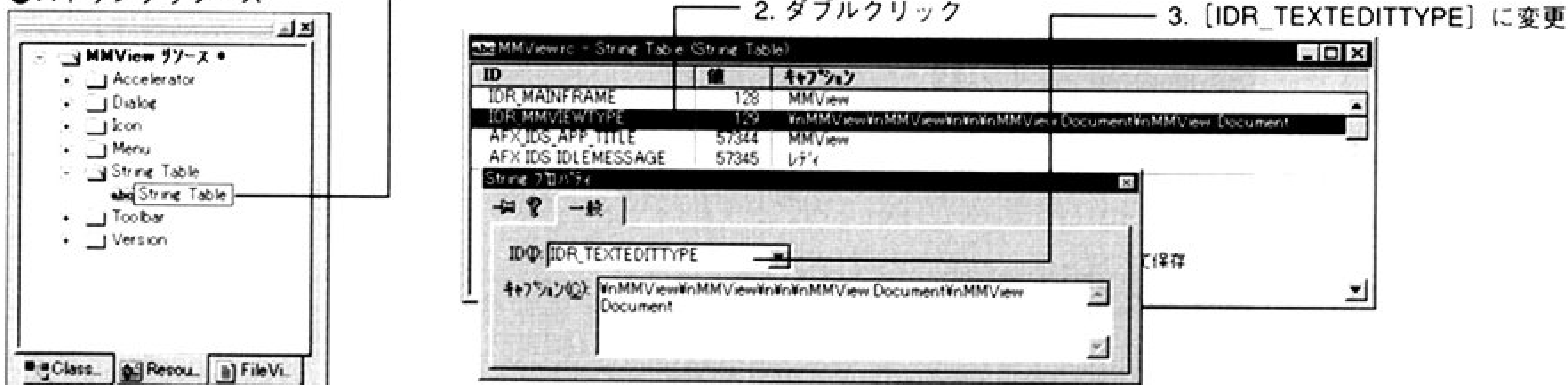


図 1-8 リソース名の書き換え

リスト 1-7 プログラム内での変更箇所（MMView.cpp）

```
CMultiDocTemplate* pDocTemplate;
pDocTemplate = new CMultiDocTemplate(
    IDR_TEXTEDITTYPE,                   // ここを変更
    RUNTIME_CLASS(CTextEditDoc),
    RUNTIME_CLASS(CChildFrame), // カスタム MDI 子フレーム
    RUNTIME_CLASS(CEditView));  // CTextEditView を CEditView に変更
AddDocTemplate(pDocTemplate);
```

ID 名の変更が済んだら、ドキュメントテンプレートに登録されるリソースを編集して、テキストエディタにふさわしいリソースに書き換えます。

● ストリングテーブルの編集

まずはストリングテーブルの編集から始めましょう。プロジェクトワークスペースウィンドウで［String Table］フォルダをダブルクリックしてストリングテーブルリソースを表示させてください。すると 1 つだけ、やはり［String Table］という名前のストリングテーブルリソースが表示されるので、これをダブルクリックしてください。するとストリングテーブル

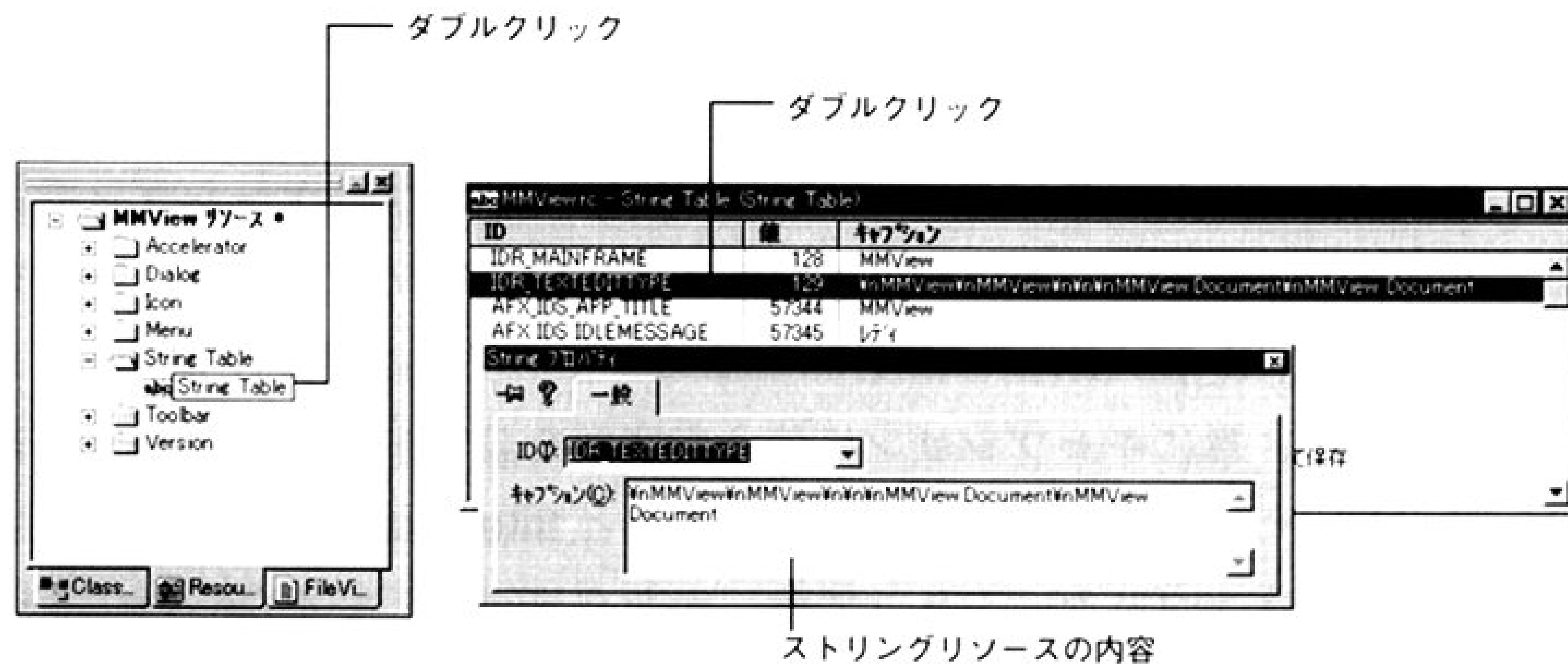


図 1-9 ストリングエディタ

エディタが起動されます。テーブルが表示されたら、その中の[IDR_TEXTEDITTYPE]をダブルクリックしてストリングエディタを開いてください(**図 1-9**)。

ストリングセグメント:0には、IDR_TEXTEDITTYPEが示す文字列が含まれています。この文字列は「¥n」によって区切られている7つのフィールドからなっています(**表 1-6**)。これらのフィールドはIDR_TEXTEDITTYPEのように、いくつかのフィールドは省略することも可能です。ただし、省略した場合には、省略したフィールドに応じて機能が制限されるので、必要なフィールドまで省略してしまわないように注意が必要です。

フィールド名	省略した場合	目的
windowTitle	空文字列	メインフレームウィンドウのキャプション(SDIアプリケーションのみ。MDIアプリケーションではIDR_MAINFRAMEストリングリソースで指定)
docName	Untitled	ドキュメントウィンドウのキャプション
fileNewName	[ファイル]ー[新規作成]コマンドで作成できなくなる	[新規作成]ダイアログで表示されるドキュメントタイプ名
filterName	[ファイル]ー[開く]コマンドで開けなくなる	[ファイルを開く]ダイアログの[ファイルの種類]に表示されるドキュメントタイプ名
filterExt	[ファイル]ー[開く]コマンドで開けなくなる	ドキュメントをファイルに保存するときに使うファイル名の拡張子
regFileTypeID	ファイルマネージャからデータファイルを指定して起動できなくなる	レジストリに登録するドキュメントタイプ名(内部管理用)
regFileTypeName	ファイルマネージャからデータファイルを指定して起動できなくなる	レジストリに登録するドキュメントタイプ名(ダイアログボックス表示用)

表 1-6 ドキュメントテンプレートに登録するストリングリソースのフォーマット

SDIアプリケーションの場合には、**windowTitle**フィールドと**docName**フィールドには、かならず適当な文字列を設定しておくべきです。そうでないと、ウィンドウのキャプションに何も表示されなくなってしまいます。ただし、MDIアプリケーションでは、その必要はなく、その代わりにIDR_MAINFRAMEストリングリソースに同じ目的で文字列を設定しておきます。IDR_MAINFRAMEストリングリソースは複数のフィールドに分割されてはいませんから、単にキャプションに使う文字列を設定するだけでかまいません。

fileNewNameフィールドと**filterName**フィールドと**filterExt**フィールドは、複数のドキュメントタイプを扱うアプリケーションでは、かならず設定してください。そうでないと、ダイアログボックスから指定したドキュメントタイプでファイルをオープンしたり、新規に作成できなくなってしまいます。

regFileTypeIDフィールドと**regFileTypeName**フィールドは、エクスプローラでデータファイルをダブルクリックすることにより、そのデータファイルのファイルタイプに関連付けられたアプリケーションを起動できるようにするのに使われますが、本書では利用しません。

ところで、AppWizardが生成したIDR_TEXTEDITTYPEには3つのフィールドしかありませんでしたが、実はfilterNameフィールドとfilterExtフィールドもAppWizardで設定することができます。設定するには、AppWizardのステップ4にある＜詳細設定＞ボタンをクリックすると表示される［詳細設定オプション］ダイアログボックスを使います。ここでドキュメントの拡張子を設定するだけです。デフォルトでは何も設定されていないので、ここでは3つしかフィールドが用意されていなかったのです。

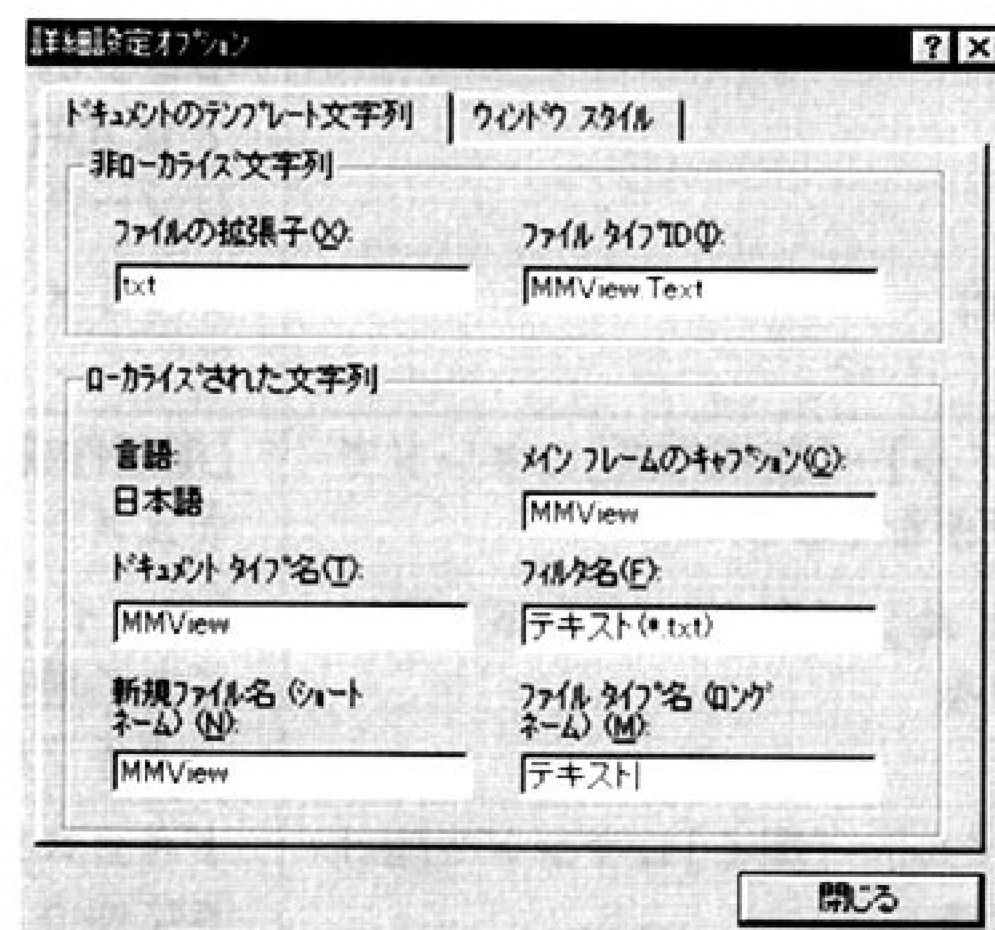


図1-10　AppWizardでドキュメントの拡張子を設定する

とりあえず、最後の2つのフィールドは省略して、次のようにIDR_TEXTEDITTYPEストリングリソースを変更しましょう。このためには、ストリングエディタでIDR_TEXTEDITTYPEをダブルクリックして、編集ウィンドウを開きます。こうしておけば、あと

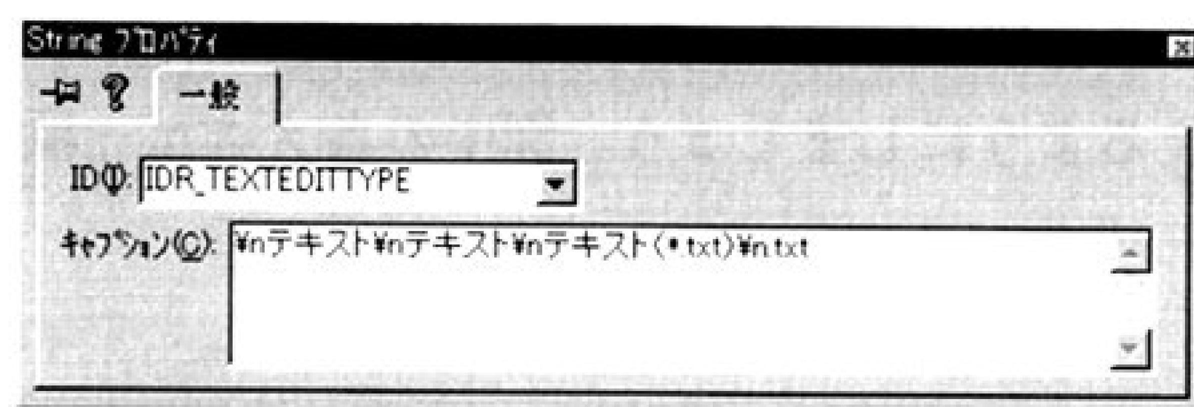


図 1-11 ストリングリソースの設定

はフレームワークから適宜参照されて、ウィンドウのキャプションやダイアログボックスのアイテムが自動的に設定されます。

メニューの追加

次にメニューリソースの編集を行います。メニューの利用方法は第2部で説明しましたが、ここでもう一度簡単に説明しておきましょう。

1. リソースエディタを使って、メニュー項目を作成し、ID を割り当てる
2. ClassWizard を使って、メニュー項目に対応するメッセージハンドラを作成する
3. メッセージハンドラを記述する

ここまでは、メニュー項目を選択したときに呼び出すような関数について何も考えていないので、メニューを追加する理由はないように思えるかもしれません。ところが、実は CEditView クラスには**表 1-7** に示すような［編集］メニューの中で使える関数とオブジェクト ID があらかじめ用意されているのです。フレームワークの中でメッセージマップも用意されているので、必要な処理はリソースエディタを使ってメニュー項目を作り、関数に対応したオブジェクト ID を振り分けるだけです。これだけでテキストエディタに必要な機能がそろってしまうのです。

オブジェクト ID	メッセージハンドラ	行う処理
ID_EDIT_CUT	OnEditCut	選択範囲の切り取り
ID_EDIT_COPY	OnEditCopy	選択範囲のクリップボードへのコピー
ID_EDIT_PASTE	OnEditPaste	クリップボードの内容の貼り付け
ID_EDIT_CLEAR	OnEditClear	選択範囲の削除
ID_EDIT_UNDO	OnEditUndo	直前の処理のやり直し
ID_EDIT_SELECT_ALL	OnEditSelectAll	すべての範囲を選択
ID_EDIT_FIND	OnEditFind	文字列の検索
ID_EDIT_REPLACE	OnEditReplace	文字列の置換
ID_EDIT_REPEAT	OnEditRepeat	直前の検索／置換の繰り返し

表 1-7 オブジェクト ID と結び付けられた操作

では、メニューエディタを使って**図 1-12** のようにメニューを書き換えて、CEditView クラスのサービスを使えるようにしましょう。図中の各メニュー項目に割り当てる ID は**表 1-8** のとおりです。

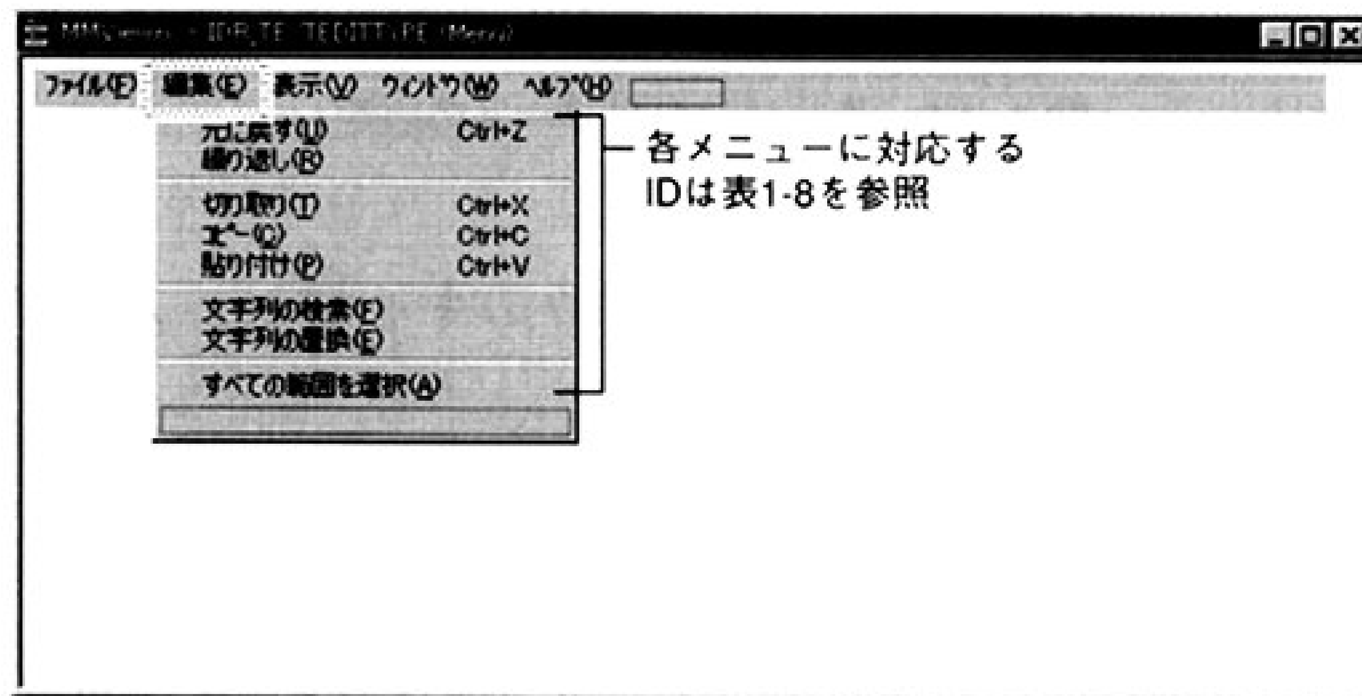


図 1-12 編集後のメニュー

メニュー項目	オブジェクト ID
元に戻す(U)	ID_EDIT_UNDO
繰り返し(R)	ID_EDIT_REPEAT
切り取り(T)	ID_EDIT_CUT
コピー(C)	ID_EDIT_COPY
貼り付け(P)	ID_EDIT_PASTE
削除(D)	ID_EDIT_CLEAR
文字列の検索(F)	ID_EDIT_FIND
文字列の置換(E)	ID_EDIT_REPLACE
すべての範囲を選択(A)	ID_EDIT_SELECT_ALL

表 1-8 メニュー項目とそのオブジェクト ID

1.9 テキストエディタ完成！

さあ、これでテキストエディタに必要な処理はすべて終わりました。さっそくコンパイル／実行してみましょう。たったあれだけのコードを追加しただけのものとは思えないほど高機能なマルチファイルテキストエディタの完成です。

ここまでに作成したプログラムのソースコードは、付属 CD-ROM の「MMView¥Step1」フォルダに格納されています。

ひとまず一区切りがついたところで、1つネタばらしをしてしまいましょう。実は本章で扱った作業は、ほとんどすべて AppWizard がやってくれることなのです。

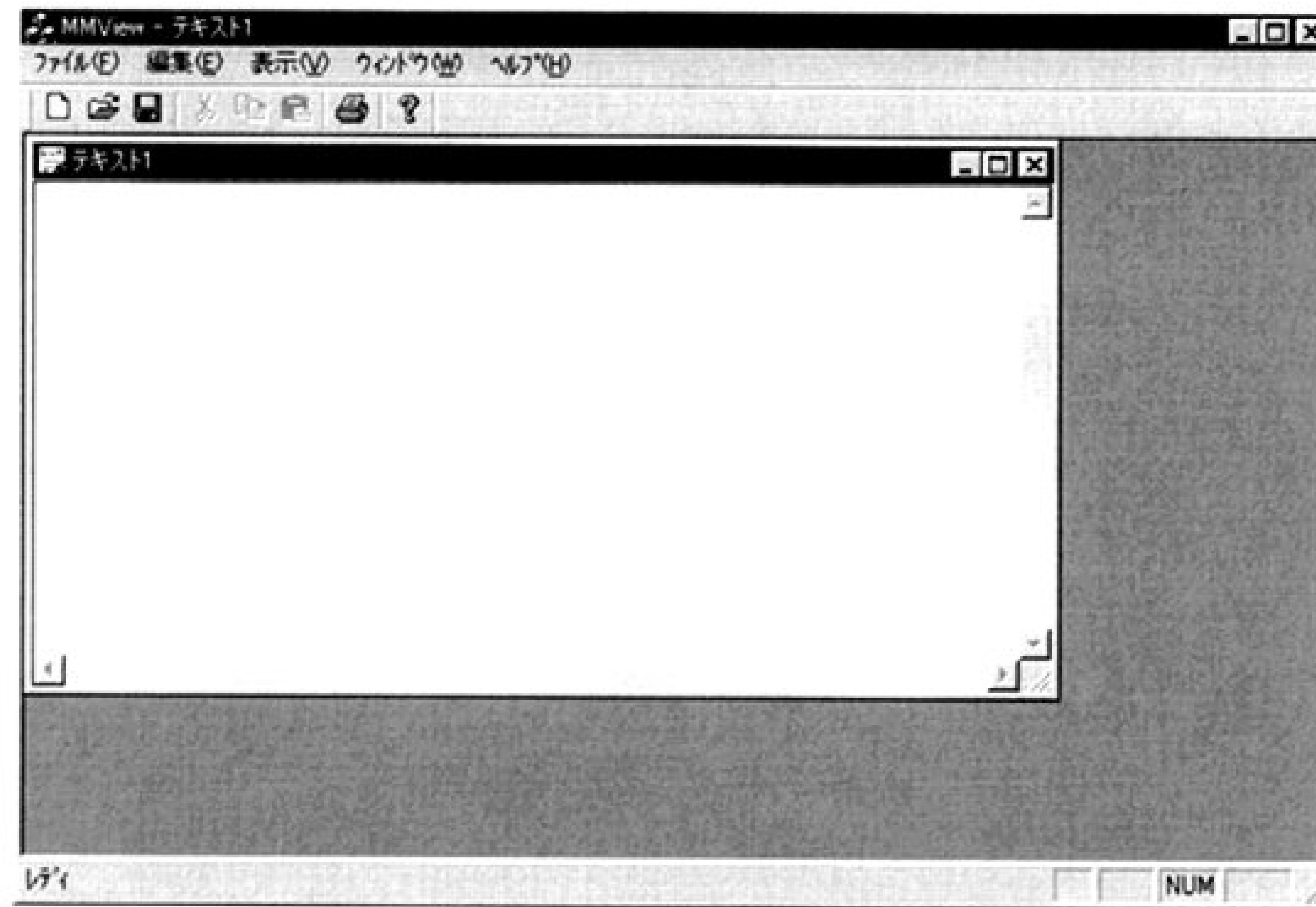


図 1-13　ここまででできたプログラムの実行画面

AppWizard のステップ 6 のダイアログボックスに、ビュークラスを選択したときにだけ現れる［基本クラス］というコンボボックスがあることに気がついた方もいるかもしれませんが、これを変更することによって、CView クラス以外のクラスを基底クラスとしてビュークラスを作ることができるのです。さらに、そこで選んだクラスに応じてちょっとしたコードも追加してくれます。たとえば、ここで CEditView クラスを選択すれば、先に追加したドキュメントクラスの Serialize メンバ関数と同じことをするコードがあらかじめ用意されます。



図 1-14　基底クラスの指定

遠回りするのも勉強のうちと納得していただいて、次章へ読み進んでください。今度は AppWizard のサポートは多くを期待できない部分ですから、ご安心ください。

2 ドローツールを作ってみよう

前章のテキストエディタはMDIアプリケーションとして作成したため、同時に複数のテキストファイルを表示したり編集したりすることができました。しかしMDIの利点はそれだけではありません。同時に複数のドキュメントタイプを扱うことができるのもMDIアプリケーションの大きな利点です（**図2-1**）。

そこで、本章では複数のドキュメントタイプが扱えるというMDIの特徴を生かして、MMViewにドローツールを追加してみます。ドローツールとは、要するにマウスを使った簡単なお絵描きプログラムです。このプログラムを作成するには、メッセージのハンドリングやGDI関数を使ったウィンドウへの図形の描画などの処理が必要になります。また、メニューやダイアログボックスを使って、ペンの太さや色を選べるようにもします。図形データはドキュメントクラスで管理し、シリアライズを使ったファイル入出力も行います。こうしてみると第2部で扱ったトピックや第3部で扱ってきたSerialize関数など非常に多くの要素が含まれるプログラムとなることが想像できます。本章では、いよいよ本格的な



図2-1　MDIアプリケーションで複数のドキュメントタイプを使う

プログラミングの作業を行うことになります。

余談になりますが、MFCでは1つのドキュメントタイプで複数のビューを利用することもできます。たとえば、テキストエディタではドキュメントとしてテキストを管理し、これを文字列としてビューに表示していますが、そのほかにも16進数のダンプリストをビューとして表示することもできます。これについては、第4部で扱うことにしましょう。

2.1 ドローツールの設計

では、最初に本章で作成するドローツールの仕様を決めましょう。扱う図形は、直線、長方形、楕円の3種類です。描画図形の選択は、**図2-2**に示すような［図形］メニューで行います。

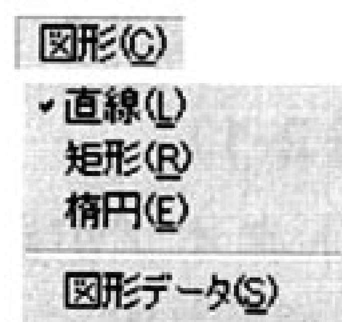


図2-2 ［図形］メニュー

［図形］メニューから［直線］を選んだ場合は、マウスで指定した2点を直線で結びます。［長方形］はマウスで指定した2点を対角線とする長方形（矩形）を描きます。［楕円］は2点を対角線とする長方形（矩形）に内接する楕円を描きます（**図2-3**）。

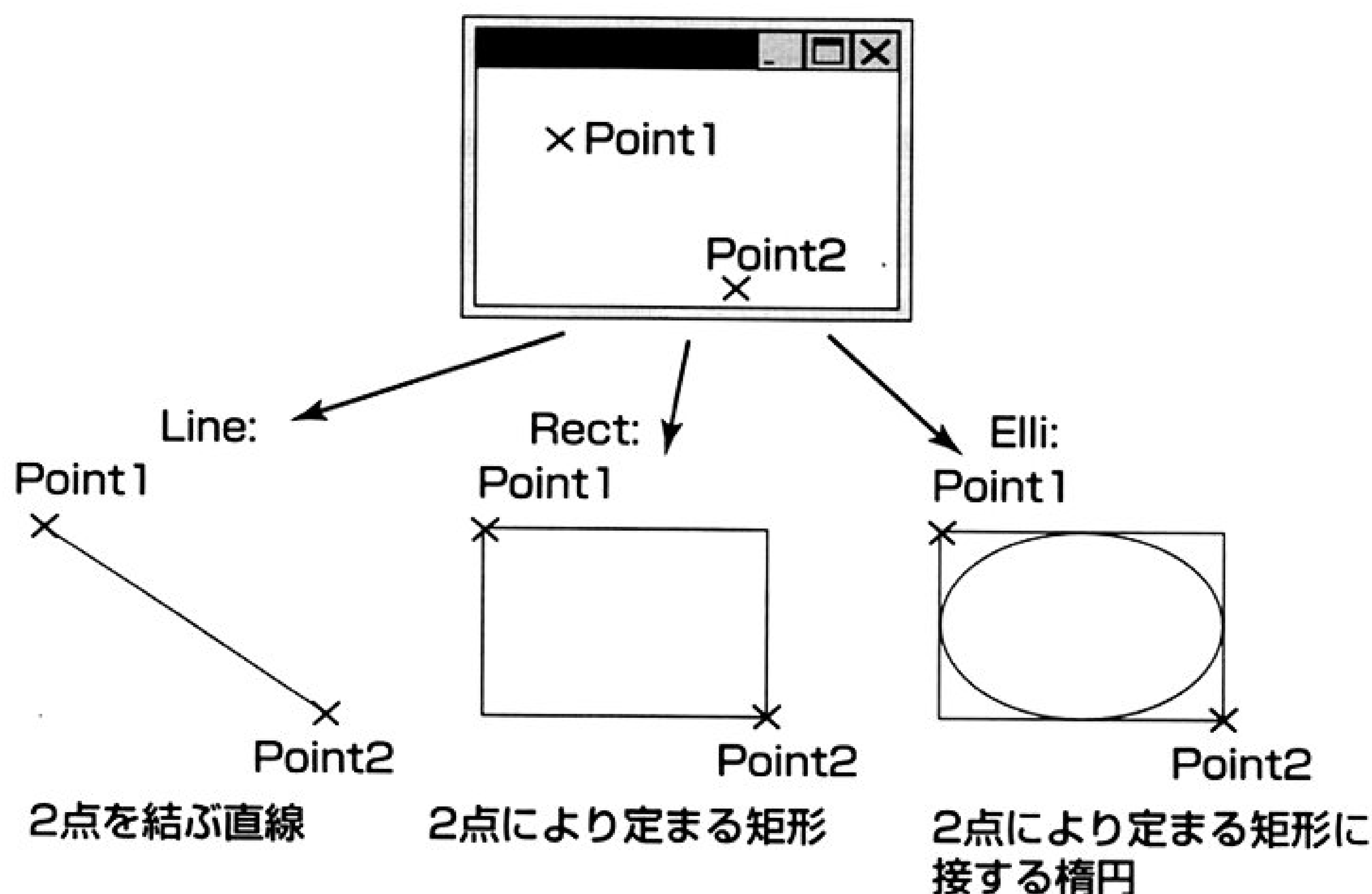


図2-3 指定した2点の位置と描画図形（直線、長方形、楕円）の関係

また［図形データ］メニューを選択すると、**図 2-4** のようなダイアログボックスが開き、ペンの太さ、ペンの色、およびブラシの色が設定できます。ここで決めたペンによって、直線、長方形の枠、楕円の枠が描画されます。ブラシは長方形と楕円の内部を塗りつぶすのに使います。ただし、［図形データの設定］ダイアログボックスを使用しての描画データの設定は DDX を使用しているだけなので、本章では触れません。実際のコードやリソースについては、付属 CD-ROM に含まれている「MMView¥Complete」フォルダの内容を参照してください。

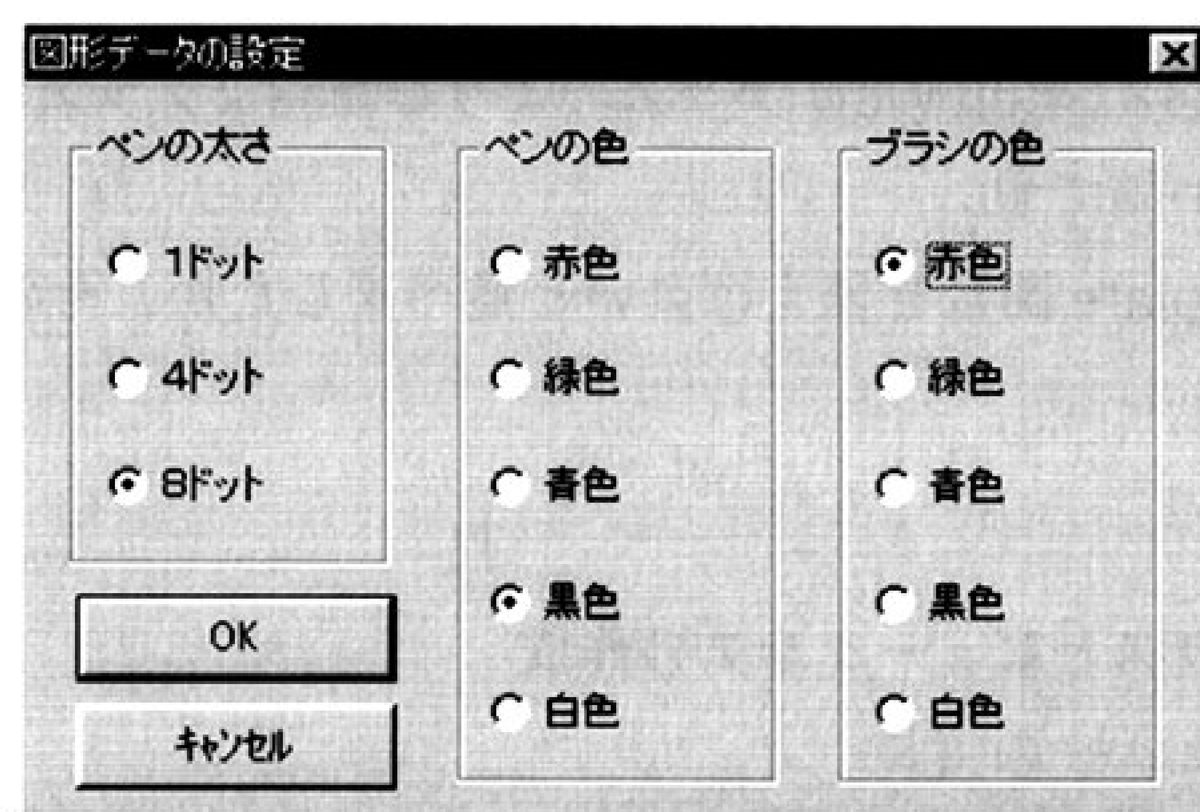


図 2-4　［図形データの設定］ダイアログボックス

描画した図形データの保存と読み込みには、［ファイル］メニューを使用します。［名前をつけて保存］や［上書き保存］を選択するとデータがファイルに保存され、［開く］を選択すると保存しておいたデータを読み込んで画面に表示することができます（**図 2-5**）。



図 2-5　［開く］ダイアログボックス

2.2 第2のドキュメントタイプ

では、このドローツールをMDIウィンドウの中にどのように表示すればよいのでしょうか? その鍵はAddDocTemplate関数にあります。この関数を使ってアプリケーションが利用するドキュメントテンプレートを登録するということはすでに述べました。つまり、AddDocTemplate関数を使って、新しくドキュメントテンプレートを登録するわけですが、そのためにはドローデータを扱うドキュメントクラスとビュークラスを新たに作成する必要があります。今度はCEditViewクラスのような便利なクラスはありません。自分で記述しなければいけないのです。

簡単にAddDocTemplate関数を使えばよいと述べましたが、そのために必要な処理はたくさんあります。

1. リソースの作成
2. ドキュメントクラスとビュークラスの作成
3. ドキュメント/ビューの実装
4. ドキュメントテンプレートへの登録

それでは1つ1つ準備をしていくことにしましょう。

2.3 IDR_DRAWTYPEリソースの作成

ドキュメントテンプレートを新しく作るためには、ドキュメントクラスとビュークラスのほかにリソースも必要です。ドキュメントテンプレートに登録されるリソースは、アイコン/メニュー/ストリングリソース/アクセラレータテーブルの4種類であることはすでに述べました。しかし、アクセラレータテーブルはかならずしも必要ではないので、ここではそれ以外の3種類のリソースを新しく作ることにしましょう。ドローデータを扱うドキュメントタイプですから、リソース名は[IDR_DRAWTYPE]とします。

リソースを新規に作成するには、プロジェクトワークスペースウィンドウにリソースの種類が一覧表示された状態で、新たに作りたいリソースの種類を選択します。そして右クリックしてショートカットメニューを表示させると、[……の挿入]という項目があるのでこれを選択します。たとえばアイコンリソースを作りたければ、[Icon]を選択したのち右クリックし、ショートカットメニューにある[Iconの挿入]を実行します。

● アイコンリソースの作成

Developer Studio にはアイコンエディタが装備されているので、これを使ってアイコン作ります。1からすべてアイコンを作るのはなかなかたいへんですし、それなりにセンスが要求されます。そんなときには Windows に付属するアイコンを利用するのも1つの手です。Developer Studio の［ファイル］－［開く］から EXE ファイルや DLL ファイルを開くと、そのファイルに含まれているアイコンなどのリソースを取り出すことができます。Windows フォルダにある Moricons.dll などには通常使われていないアイコンリソースが含まれているので、個人的に利用するプログラムであれば、こうしたアイコンを利用するのもよいでしょう（**図 2-6**）。リソース ID の値を明示的に指定したいときには「リソース ID 名＝リソース ID 値」の形式でリソース ID を指定してください。

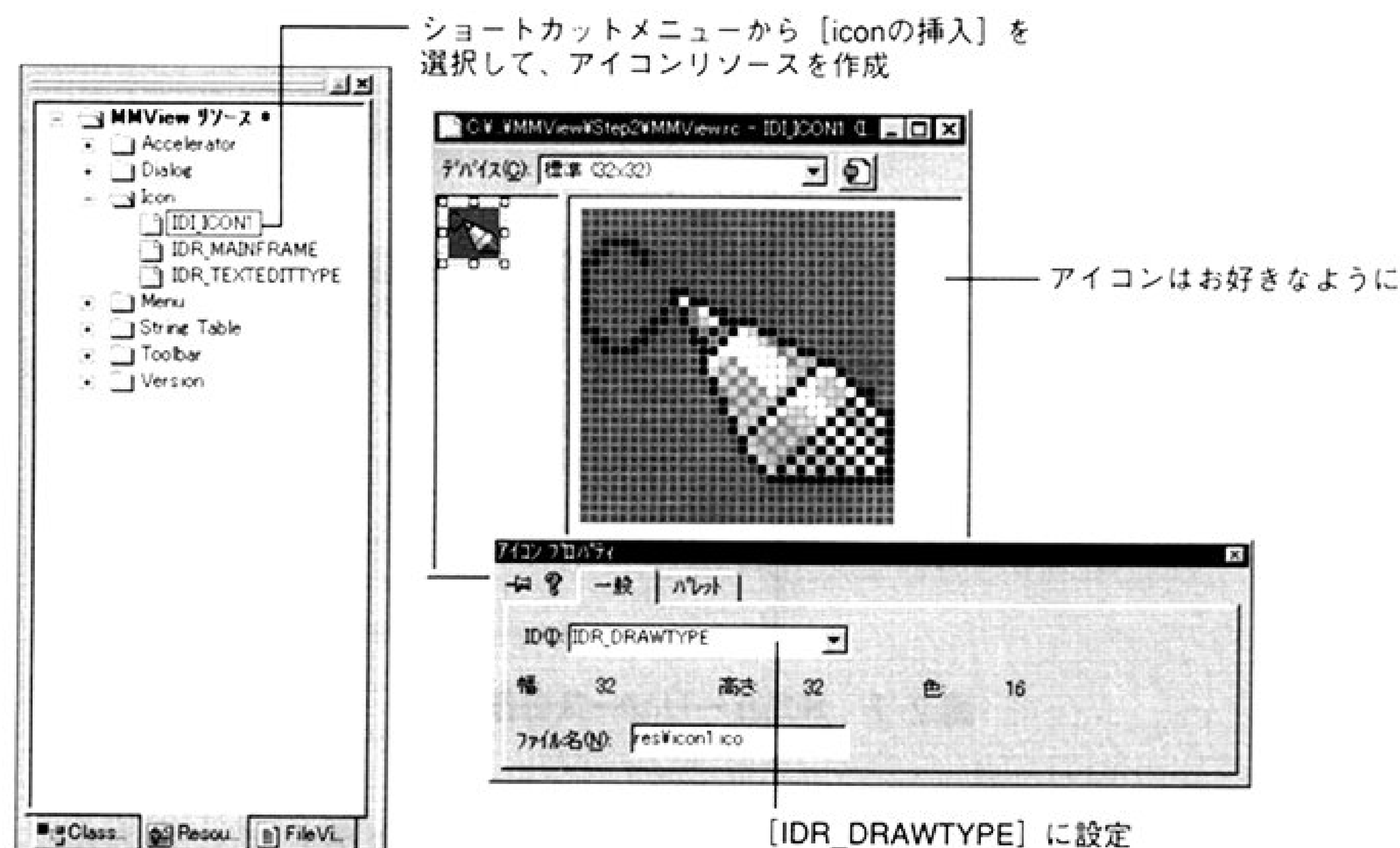


図 2-6　アイコンの設定

● メニューリソースの作成

メニューは、メニューエディタを使って作りますが、ドローツールで必要なメニューは、テキストエディタのメニューとほとんど同じです。そこで、テキストエディタのメニューリソース、「IDR_TEXTEDITTYPE」から必要な項目をコピーするのが一番てっとり早い方法です。

まずアイコンリソースと同じようにワークスペースウィンドウから新しくメニューリソースを作ります。次にテキストエディタのメニュー（IDR_TEXTEDITTYPE）を開き、作成したばかりのメニューと並べて画面に表示させてください。ここで IDR_TEXTEDITTYPE メニューの［ファイル］メニューを Ctrl を押しながらドラッグし、IDR_DRAWTYPE メニューの上にドロップします。すると［ファイル］メニューのすべてのメニュー項目をコ

ピーできます。または、マウスの右ボタンでドラッグ&ドロップをしてもかまいません。この場合、ドロップした時点でショートカットメニューが表示されるので、[コピー]を選択します。同じ要領で、[表示]メニューと[ウィンドウ]メニューと[ヘルプ]メニューもコピーしてください。最後に[図形]メニューを追加すればメニューは完成します(**図2-7**)。

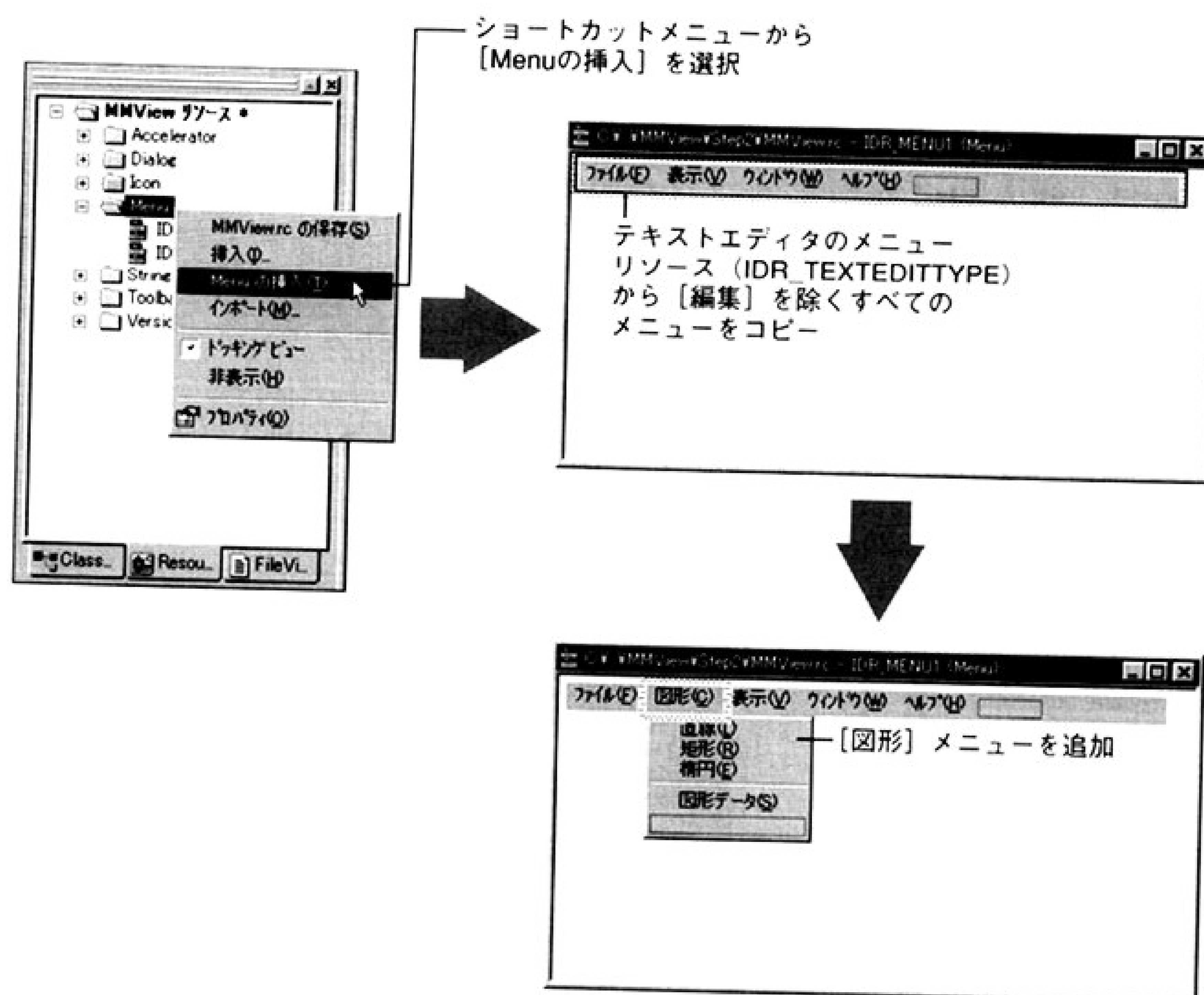


図2-7 メニューリソースの作成

作成したメニューのIDはIDR_DRAWTYPEに忘れずに変更してください。また、[図形]メニューの各メニュー項目のID名については**表2-1**のようにしておいてください。

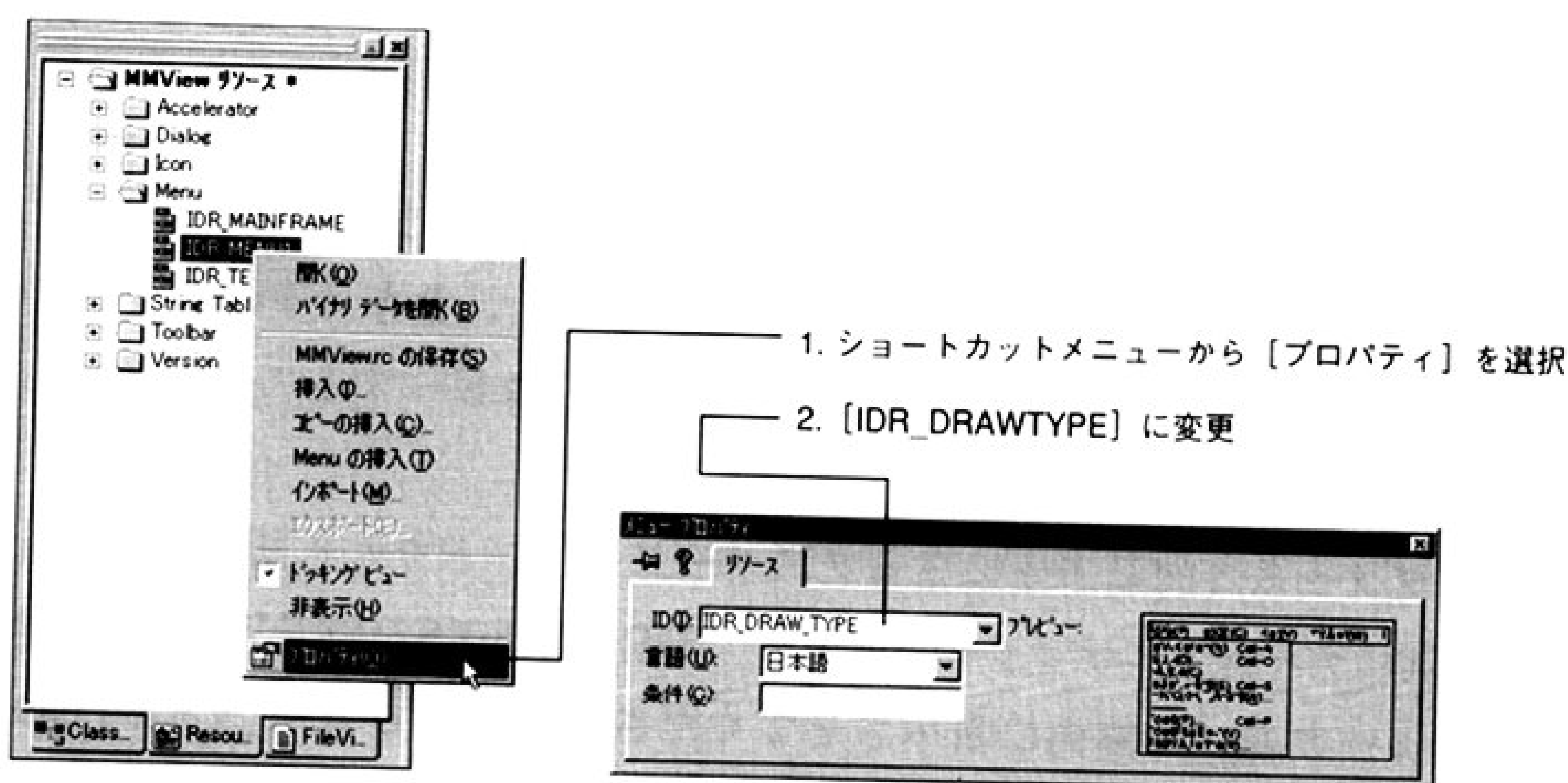


図2-8 リソースIDの設定

メニュー項目	ID 名
直線(L)	ID_STYLE_LINE
矩形(R)	ID_STYLE_RECTANGLE
楕円(E)	ID_STYLE_ELLIPSE
図形データ(S)	ID_STYLE_DATASET

表 2-1 [図形] メニューのメニュー項目のオブジェクト ID

● ストリングリソースの作成

ストリングリソースは、ストリングエディタを利用して作成します。そのためにはまず新しいストリングリソースのための場所を空けなければなりません。今度はストリングリソースを新しく作るのではなく、既存のストリングリソーステーブルに新しいエントリを追加するという作業を行います。ワークスペースウィンドウから [String Table] をダブルクリックして、ストリングエディタを起動してください。ストリングテーブルが開いたら、その中から [IDR_TEXTEDITTYPE] を選択します。そして Insert を押すか、右クリックしてショートカットメニューから[ストリングの新規作成]を実行します。すると、IDR_TEXTEDITTYPE リソースの下にスペースができるので、ここに IDR_DRAWTYPE ストリングリソースを設定します(**図 2-9**)。

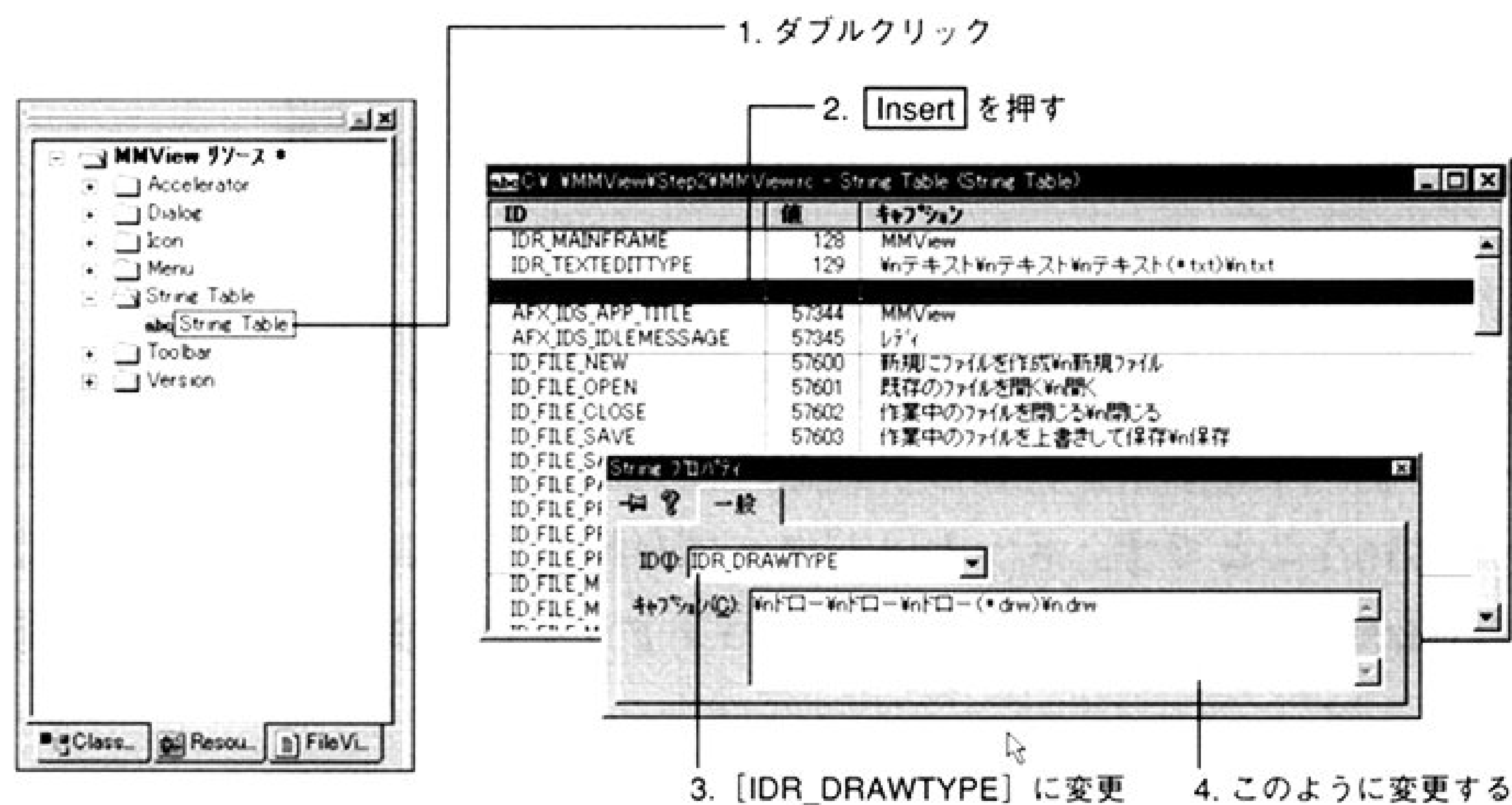


図 2-9 ストリングの設定

以上でリソースの作成は終了です。次に、これらのリソースを利用してドローデータを管理／表示するためのビュークラスとドキュメントクラスを作成します。

2.4 派生クラスの作成

では次に、新しくドキュメントクラスとビュークラスを作ることにしましょう。メニューから［表示］－［ClassWizard］コマンドを実行して、ClassWizardを起動してください。派生クラスを生成するためには、ClassWizardの＜クラスの追加＞ボタンを使います。このボタンをクリックするとドロップダウンメニューが開くので、ここから［新規］を選択します。すると、第2部でも使用した［クラスの新規作成］ダイアログボックスが開きます。［クラス名］に新規に作成するクラスの名前を指定して、そのクラスの基底クラスは［基本クラス］に表示されるリストから選択します。

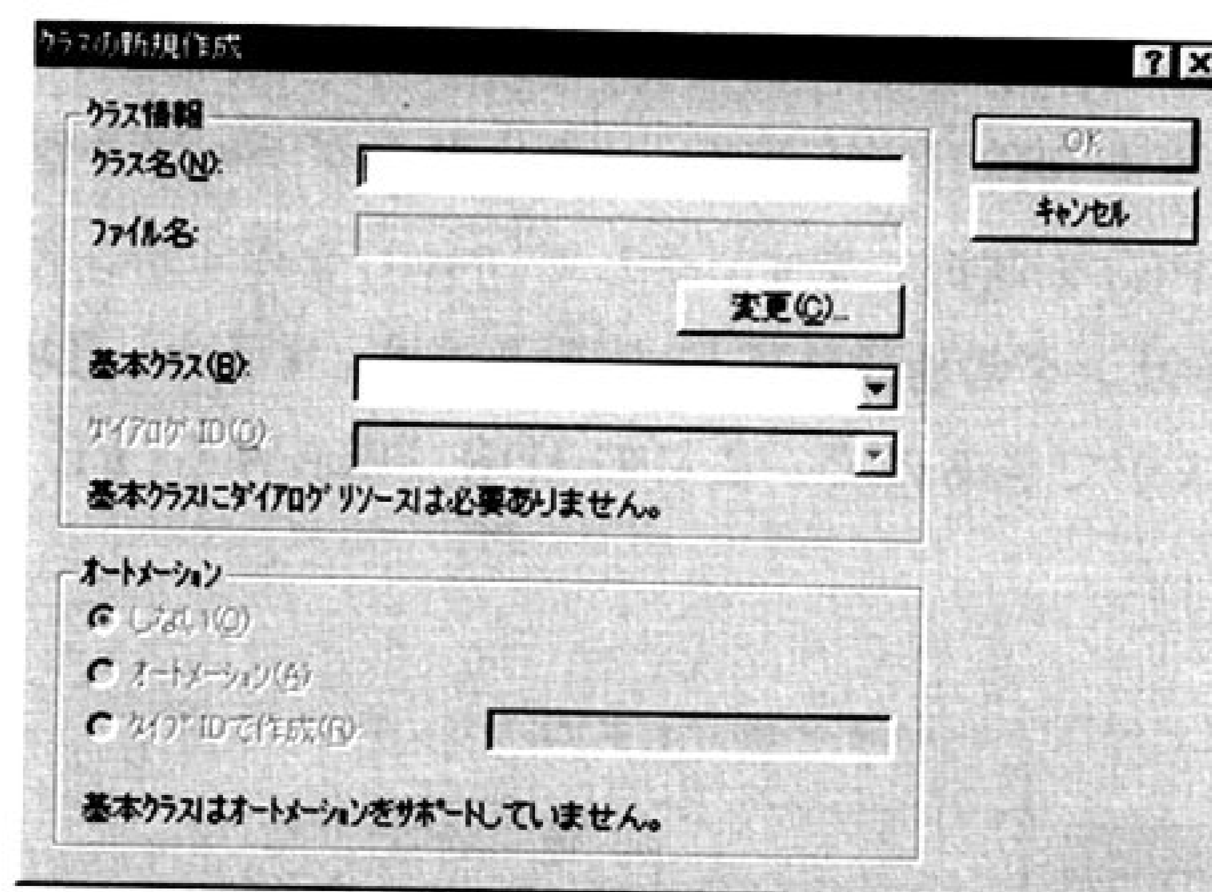


図2-10　新規クラスの作成

ClassWizardを使えば、約50種類ものクラスから基底クラスを選んで（**表2-2**）、派生クラスを作ることができます。これらのクラスに共通していることは、メッセージマップを持っているということです。メッセージマップを持っているということは、これらのクラスはClassWizardを使ってそのメッセージハンドラを管理することができるということです。

ところで、メッセージマップの管理を行う必要のないクラスを作るのであれば、ClassWizardを使って派生クラスを作る必要はありません。ClassWizardを使って派生クラスを作ると、単なるクラスの定義だけでなく、あとでメッセージハンドラを管理するために必要なメッセージマップや、基底クラスに応じた初期設定コードなどが追加されます。こうしたサービスがあってこそのClassWizardなのです。仮にCViewクラスの派生クラスを手作業で作り、メッセージマップを記述しなかった場合には、たとえCViewクラスの派生クラスといえどもClassWizardを使ったメッセージハンドラの管理はできない点に注意をしてください。

基底クラス	用途
CDialog	ダイアログボックス
CDocument	ドキュメントクラス
CFormView	ダイアログボックスのようなビュークラス
CFrameWnd	SDI アプリケーションのメインフレームウィンドウ
CMDIChildWnd	MDI アプリケーションの子ウィンドウ
CScrollView	スクロールバーのついたビュークラス
CView	汎用的なビュークラス
generic CWnd	一般的なウィンドウ
splitter	複数の表示領域を持つ子ウィンドウ

表 2-2　基底クラスになりうるクラスの一部

ここでは、CDocument クラスを基底クラスにした CDrawDoc クラスと、CView クラスを基底クラスにした CDrawView クラスを作ることにしましょう。クラス名とその基底クラス、およびクラスの定義を記述するファイルのファイル名は**表 2-3** のように設定することにします。

クラス名	基底クラス	インプリメントファイル	ヘッダファイル
CDrawDoc	CDocument	DrawDoc.cpp	DrawDoc.h
CDrawView	CView	DrawView.cpp	DrawView.h

表 2-3　作成するクラスの設定

必要な項目を設定したら、<OK>ボタンをクリックしてください。すると指定したクラスの定義を含んだファイルが作られます。

ClassWizard が生成したコードには、クラスの定義だけではなく、ほとんどの場合オーバーライドする必要がある仮想関数があらかじめ用意されています。たとえば、CView クラスの派生クラスには OnDraw 関数が、CDocument クラスの派生クラスには Serialize 関数がそれぞれ用意されます。これら以外の基底クラスを選んだ場合には、やはりそれぞれのクラスで必要になる仮想関数が用意されます。

2.5 ビューとドキュメントの実装

派生クラスの作成が終わりましたから、これからはドキュメントクラスとビュークラスの実装に移ります。ドキュメントクラスでは図形データを管理するクラスの作成などを、ビュークラスではクライアント領域へのユーザーのマウスの入力を受け付けたり、描く図形をメニューによって選択できるようにします。

まずはドキュメントクラスの実装から始めることにしましょう。

図形データを管理する

このドローツールで扱う図形(直線、長方形、楕円)は**図 2-3** にも示しましたが、どれも2点を指定するだけでその位置とサイズが決まります。したがって、以下の6つのパラメータを用意すれば、図形に関するすべての要素を表せます。

- **図形の種類(直線/長方形/楕円)**
- **ペンの太さ**
- **ペンの色**
- **ブラシの色(直線を描画する場合は無視する)**
- **第1の点の座標**
- **第2の点の座標**

そこで**リスト 2-1** に示すような CShape クラスを定義して図形データを管理することにしましょう。また図形の種類を表すために、STYLE 型というデータ型も宣言します。STYLE 型は、Line/Rect/Elli の3つの値のいずれかを取る列挙型です。

図形のパラメータは、m_style から m_point2 までの6つのプライベートなメンバ変数で表します。プライベートなメンバとは、そのメンバが属するクラスのメンバからしかアクセスできないメンバのことです(詳細は Appendix A を参照)。この場合は、CShape クラスのメンバ関数しかこれらのメンバ変数の値を変更したり、その値を読み取ることができないのです。

ただし、「friend class CDrawDoc;」という宣言がありますから、CDrawDoc クラスだけは例外的に CShape クラスのプライベートメンバにもアクセスすることができます。これは、CDrawDoc クラス内でデータを記憶するために CShape クラスを利用するので、プライベートメンバにもアクセスできると何かと都合のいいことがあるからです。このような場合、CDrawDoc クラスを「CShape クラスのフレンドクラス」といいます。ドキュメントクラス内で管理するデータについて自分でクラスを作成するときには、ドキュメントクラスをそのクラスのフレンドクラスにしておくといいことが結構ありますから、覚えてお

いてください。CShapeクラスのメンバ関数はメンバ変数の値を返したり、引数に指定された値をメンバ変数に代入したりするだけのものですから、クラス定義の中にメンバ関数の定義を直接含めてしまうことにします。

リスト2-1 CShapeクラスの定義(DrawDoc.h)

```
///////////////////////////////////////
// 列挙型 STYLE

enum STYLE { Line, Rect, Elli };

///////////////////////////////////////
// CShape クラス

class CShape
{
  friend class CDrawDoc;  //フレンドクラスの宣言

private:
  STYLE     m_style;       //図形の種類
  int       m_penwidth;    //ペンの太さ
  COLORREF  m_pencolor;    //ペンの色
  COLORREF  m_brushcolor;  //ブラシの色
  CPoint    m_point1;      //第1の点の座標
  CPoint    m_point2;      //第2の点の座標

public:
  void    SetStyle(STYLE s)          { m_style = s; }
  void    SetPenWidth(int w)         { m_penwidth = w; }
  void    SetPenColor(COLORREF c)    { m_pencolor = c; }
  void    SetBrushColor(COLORREF c)  { m_brushcolor = c; }
  void    SetPoint1(CPoint pt)       { m_point1 = pt; }
  void    SetPoint2(CPoint pt)       { m_point2 = pt; }

  STYLE     GetStyle()      { return m_style; }
  int       GetPenWidth()   { return m_penwidth; }
  COLORREF  GetPenColor()   { return m_pencolor; }
  COLORREF  GetBrushColor() { return m_brushcolor; }
  CPoint    GetPoint1()     { return m_point1; }
  CPoint    GetPoint2()     { return m_point2; }
};
```

CShapeクラスのようにプログラマがまったく新規にクラスを定義する場合には、ClassWizardの助力は仰げません。つまりクラス定義のコードは全部自分の手で入力する必要があるということです。CShapeクラスはドキュメントクラス内で利用するものですから、DrawDoc.hの中で定義すればよいでしょう。**リスト2-1**に示したコードはDrawDoc.hファイルのCDrawDocクラスの定義の直前に挿入してください。ヘッダファイル(DrawDoc.h)

を開くには、ワークスペースウィンドウの［ClassView］ページで［CDrawDoc］をダブルクリックします。

次に、CShapeクラスをドキュメントクラスCDrawDocの中に組み込みます。といってもたいして難しい作業ではなく、**リスト2-2**のアミがかかった部分のようにCDrawDocクラスにCShapeクラスのメンバ変数を1つ用意し、そのメンバ変数を操作する一連のメンバ関数を定義するだけです。

リスト2-2　CDrawDocクラスにメンバ変数とメンバ関数を追加（DrawDoc.h）

```
class CDrawDoc : public CDocument
{
protected:
  CDrawDoc();        // 動的生成に使用されるプロテクト コンストラクタ

// アトリビュート
public:
  CShape  m_shape;     // 現在描画中の図形を表す

  void    SetStyle(STYLE s)          { m_shape.SetStyle(s); }
  void    SetPenWidth(int w)         { m_shape.SetPenWidth(w); }
  void    SetPenColor(COLORREF c)    { m_shape.SetPenColor(c); }
  void    SetBrushColor(COLORREF c)  { m_shape.SetBrushColor(c); }
  void    SetPoint1(CPoint pt)       { m_shape.SetPoint1(pt); }
  void    SetPoint2(CPoint pt)       { m_shape.SetPoint2(pt); }

  STYLE    GetStyle()              { return m_shape.GetStyle(); }
  int      GetPenWidth()           { return m_shape.GetPenWidth(); }
  COLORREF  GetPenColor()          { return m_shape.GetPenColor(); }
  COLORREF  GetBrushColor()        { return m_shape.GetBrushColor(); }
  CPoint    GetPoint1()            { return m_shape.GetPoint1(); }
  CPoint    GetPoint2()            { return m_shape.GetPoint2(); }

// インプリメンテーション
protected:
  virtual ~CDrawDoc();
  ...略
};
```

CDrawDocクラスのコンストラクタの定義

これで、図形データを処理するための準備ができました。が、このままではドキュメントクラスのオブジェクトが作成された直後の状態が不安定です。直線を描くのか？ ペンの色や太さは？ ブラシの色は？ といった要素が何も決まっていません。そこで、ドキュメントクラスのコンストラクタ（CDrawDoc::CDrawDoc関数）でこれらの初期設定を行ってしまうことにしましょう。もちろん、CShapeクラスのコンストラクタを利用してもかまいませんが、図形データの初期化が必要なのは、新しいドキュメントを作るときだけなので、

本プログラムではCDrawDocクラスのコンストラクタで初期設定を行うことにしました。コンストラクタのリストは**リスト2-3**に示します。

リスト2-3 CDrawDoc::CDrawDoc関数(DrawDoc.cpp)

```
CDrawDoc::CDrawDoc()
{
  m_shape.SetStyle(Line);                    // 初期設定では直線を描く
  m_shape.SetPenWidth(3);                    // ペン幅は3
  m_shape.SetPenColor(RGB(0,0,0));           // ペンの色は黒
  m_shape.SetBrushColor(RGB(255,255,255));   // ブラシの色は白
}
```

ここで設定された図形データのうち、描く図形の種類は[図形]メニューから[直線]、[矩形]、[楕円]の各メニュー項目を選択して、その他のデータは[図形データ]メニューで表示される[ペンとブラシの設定]ダイアログボックスを使って変更することができます。

[ペンとブラシの設定]ダイアログボックスでは、実際にはDDXを利用して、ダイアログボックスでのユーザーの選択をCShapeクラスのメンバ変数、m_shapeの各メンバに記憶させているだけです。ダイアログボックスを作成してクラスを登録し、DDX変数を作成する手順は第2部の説明の繰り返しになるので、ここでは行いません。実際に、どんなことをしているかは、先ほどもいったように付属CD-ROMの「MMView¥Complete」フォルダに収められているプログラムリストを参考にしてください。

したがって、ここまでの説明どおりにMMViewを作成しただけでは、[ペンとブラシの設定]ダイアログボックスを利用しての図形の描画スタイルの変更は行えませんから気をつけてください。

●ビュークラスの実装

次にビュークラスの実装を行います。実装の多くはユーザーのクライアント領域/メニューを使った操作に対応するメッセージハンドラの作成です。そこで、まず図形スタイルを指定する[直線]、[長方形]、[楕円]の3つのメニューコマンドに対してメッセージハンドラを作成します(**リスト2-4**)。CDrawViewクラスのオブジェクトは、これらのハンドラを実行することによって、自分のドキュメントの中に含まれる描画スタイルの設定を変更します。

MDIアプリケーションでは、メニューからのメッセージは、その時点でアクティブな子ウィンドウに送られます。したがって、CDrawViewクラスのウィンドウが複数表示されていても、メニューによって描画スタイルが変更されるのは、アクティブなウィンドウが管理しているドキュメント(とそのドキュメントに関連付けられているビュー)についてのみに限られることに注意してください。

クラス名	オブジェクト ID	メッセージ	メッセージハンドラ
CDrawView	ID_STYLE_LINE	COMMAND	OnStyleLine
CDrawView	ID_STYLE_RECTANGLE	COMMAND	OnStyleRectangle
CDrawView	ID_STYLE_ELLIPSE	COMMAND	OnStyleEllipse

表2-4 描画スタイルを変更するメッセージハンドラ

リスト2-4 描画スタイル変更用メッセージハンドラの実装(DrawView.cpp)

```
void CDrawView::OnStyleLine()
{
  GetDocument()->SetStyle(Line);     // 描画スタイルを Line にする
}

void CDrawView::OnStyleRectangle()
{
  GetDocument()->SetStyle(Rect);     // 描画スタイルを Rect にする
}

void CDrawView::OnStyleEllipse()
{
  GetDocument()->SetStyle(Elli);     // 描画スタイルを Elli にする
}
```

また、この3つのメニューのうち、現在選択されている描画スタイルにチェックマークを付けるため、**リスト2-5**に示すUPDATE_COMMAND_UIメッセージハンドラを作成します。メッセージハンドラの指定は**表2-5**を参考にしてください。

クラス名	オブジェクト ID	メッセージ	メッセージハンドラ
CDrawView	ID_STYLE_LINE	UPDATE_COMMAND_UI	OnUpdateStyleLine
CDrawView	ID_STYLE_RECTANGLE	UPDATE_COMMAND_UI	OnUpdateStyleRectangle
CDrawView	ID_STYLE_ELLIPSE	UPDATE_COMMAND_UI	OnUpdateStyleEllipse

表2-5 チェックマークを付けるメッセージハンドラ

リスト2-5 UPDATE_COMMAND_UIメッセージのハンドラ(DrawView.cpp)

```
void CDrawView::OnUpdateStyleLine(CCmdUI* pCmdUI)
{
  pCmdUI->SetCheck(GetDocument()->GetStyle() == Line);
}

void CDrawView::OnUpdateStyleRectangle(CCmdUI* pCmdUI)
```

```
{
  pCmdUI->SetCheck(GetDocument()->GetStyle() == Rect);
}

void CDrawView::OnUpdateStyleEllipse(CCmdUI* pCmdUI)
{
  pCmdUI->SetCheck(GetDocument()->GetStyle() == Elli);
}
```

CDrawViewクラスのクライアント領域でマウスが操作された場合は、**リスト2-6**に示す3つのメッセージハンドラが呼び出されて、画面に図形を表示します。ここでは、OnDraw関数を使わずに、OnMouseMove関数とOnLButtonUp関数の中で図形の描画をしています。このように、すべてをOnDraw関数でまかなわずに、それぞれのルーチンでCWnd::GetDC関数を使ってデバイスコンテキストを取得して画面への描画を行うことも可能です。

マウスの左ボタンが押されると、図形の始点が決定し、そのあとボタンが離されるまでOnMouseMove関数の中で図形の始点からマウスカーソルまでどんな図形になるのかを知らせるためにダミーの図形が描画されます。このときに、XORのマスクを利用して、直前のOnMouseMove関数で描いたダミーの図形を消去していることに注意してください。マウスの左ボタンが離されると、図形の終点も決定し、画面に図形が描画されます。

クラス名	オブジェクトID	メッセージ	メッセージハンドラ
CDrawView	なし(CDrawViewを選択)	WM_LBUTTONDOWN	OnLButtonDown
CDrawView	なし(CDrawViewを選択)	WM_MOUSEMOVE	OnMouseMove
CDrawView	なし(CDrawViewを選択)	WM_LBUTTONUP	OnLButtonUp

表2-6 メッセージハンドラの指定

リスト2-6 図形を描画するためのメッセージハンドラ(DrawView.cpp)

```
void CDrawView::OnLButtonDown(UINT nFlags, CPoint point)
{
  CDrawDoc* pDoc = GetDocument();

  pDoc->SetPoint1(point);
  pDoc->SetPoint2(point);
  SetCapture();              // マウスのキャプチャーを開始
}

void CDrawView::OnMouseMove(UINT nFlags, CPoint point)
{
```

```
  CDrawDoc* pDoc = GetDocument();
  CDC*  pDC;
  CPen  dashPen(PS_DASH, 1, RGB(255, 255, 255));
  CPen* pOldPen;
  CPoint  pt1 = pDoc->GetPoint1();
  CPoint  pt2 = pDoc->GetPoint2();

  if (!GetCapture())  // マウスがキャプチャーされていないなら(マウス
    return;           // ボタンが押されていないなら)何もしない

  pDC = GetDC();                               // デバイスコンテキストの取得
  pDC->SetROP2(R2_XORPEN);                     // XORのマスクをかける
  pDC->SelectStockObject(NULL_BRUSH);          // ヌルブラシを選択
  pOldPen = pDC->SelectObject(&dashPen);       // ダッシュペンを選択

  switch (pDoc->GetStyle()) {
  case Line:
    pDC->MoveTo(pt1);   // 直前に引いた直線の始点に移動して、もう一度同じ
    pDC->LineTo(pt2);   // 直線を引くとXORのマスクのためにその線が消える
    pDC->MoveTo(pt1);   // 再び直線の始点に移動して
    pDC->LineTo(point); // マウスカーソルの位置まで直線を引く
    break;
  case Rect:
    pDC->Rectangle(pt1.x, pt1.y, pt2.x, pt2.y);      // 直前に描いた矩形の消去
    pDC->Rectangle(pt1.x, pt1.y, point.x, point.y);  // 矩形の描画
    break;
  case Elli:
    pDC->Ellipse(pt1.x, pt1.y, pt2.x, pt2.y);        // 直前に描いた楕円の消去
    pDC->Ellipse(pt1.x, pt1.y, point.x, point.y);    // 楕円の描画
    break;
  }
  pDoc->SetPoint2(point); // 現在のマウスカーソル位置を記憶

  pDC->SelectObject(pOldPen);
  ReleaseDC(pDC);          // デバイスコンテキストの解放
}

void CDrawView::OnLButtonUp(UINT nFlags, CPoint point)
{
  CDrawDoc* pDoc = GetDocument();
  CDC*  pDC;
  CPen  dashPen(PS_DASH, 1, RGB(255, 255, 255));
  CPen  drawPen(PS_INSIDEFRAME, pDoc->GetPenWidth(), pDoc->GetPenColor());
  CPen* pOldPen;
  CBrush  Brush(pDoc->GetBrushColor());
  CPoint  pt1 = pDoc->GetPoint1();
  CPoint  pt2 = pDoc->GetPoint2();

  pDC = GetDC();                           // デバイスコンテキストの取得
```

```
    pDC->SetROP2(R2_XORPEN);                    // XOR のマスク
    pDC->SelectStockObject(NULL_BRUSH);
    pOldPen = pDC->SelectObject(&dashPen);

    switch (pDoc->GetStyle()) {
    case Line:
      pDC->MoveTo(pt1);                         // 直線の消去
      pDC->LineTo(pt2);
      pDC->SelectObject(&drawPen);              // 描画用ペンを選択
      pDC->SetROP2(R2_COPYPEN);                 // 直線の色はペンの色
      pDC->MoveTo(pt1);                         // 直線の描画
      pDC->LineTo(point);
      break;
    case Rect:
      pDC->Rectangle(pt1.x, pt1.y, pt2.x, pt2.y); // 矩形の消去
      pDC->SelectObject(&Brush);                    // 描画用ブラシの選択
      pDC->SelectObject(&drawPen);                  // 描画用ペンの選択
      pDC->SetROP2(R2_COPYPEN);                     // 枠線の色はペンの色
      pDC->Rectangle(pt1.x, pt1.y, point.x, point.y); // 矩形の描画
      break;
    case Elli:
      pDC->Ellipse(pt1.x, pt1.y, pt2.x, pt2.y);     // 楕円の消去
      pDC->SelectObject(&Brush);                    // 描画用ブラシの選択
      pDC->SelectObject(&drawPen);                  // 描画用ペンの選択
      pDC->SetROP2(R2_COPYPEN);                     // 枠線の色はペンの色
      pDC->Ellipse(pt1.x, pt1.y, point.x, point.y); // 楕円の描画
      break;
    }

    pDC->SelectObject(pOldPen);
    ReleaseDC(pDC);    // デバイスコンテキストの解放
    ReleaseCapture(); // マウスのキャプチャーの終了
  }
```

CDrawView::GetDocument 関数の定義

今までの関数中で CDrawDoc クラスのオブジェクトを得るために何気なく使っていた GetDocument 関数ですが、CDrawView クラスではプログラマがこの関数を用意しなければいけません。

AppWizard でスケルトンを作成する場合は、ドキュメントクラスとビュークラスはペアとなることを前提として作られるため、両者の関係を反映するようなコードが生成されました。しかし CDrawView クラスと CDrawDoc クラスは、それぞれ個別に ClassWizard で作ったクラスです。この場合、ClassWizard は CDrawView クラスと CDrawDoc クラスが対応していることを知らないので、両者にかかわる GetDocument 関数は生成できないのです。

そこで、CDrawViewクラスにも、**リスト2-7**のようなGetDocument関数を追加します。あるクラスにメンバ関数やメンバ変数を追加する場合には、ワークスペースウィンドウの［ClassView］ページで、対象となるクラスを右クリックし、ショートカットメニューから［メンバ関数の追加］あるいは［メンバ変数の追加］を選択すると簡単です。とくに、この方法でメンバ関数を挿入した場合、定義ファイル（.hファイル）と実装ファイル（.cppファイル）の両者に対して変更がなされるので、プログラマの手を煩わせることがありません。

ここでは、**図2-11**に示すように［関数の型］には［CDrawDoc*］を、［関数の宣言］には［GetDocument()］を指定して<OK>ボタンをクリックします。このとき、［アクセス制御］は［public］のまま、［static］と［virtual］もチェックをする必要はありません。無論、追加したりオーバーライドする関数に応じて、これらは適宜指定する必要があります。関数を追加したら、**リスト2-7**に示すコードを記述してください。

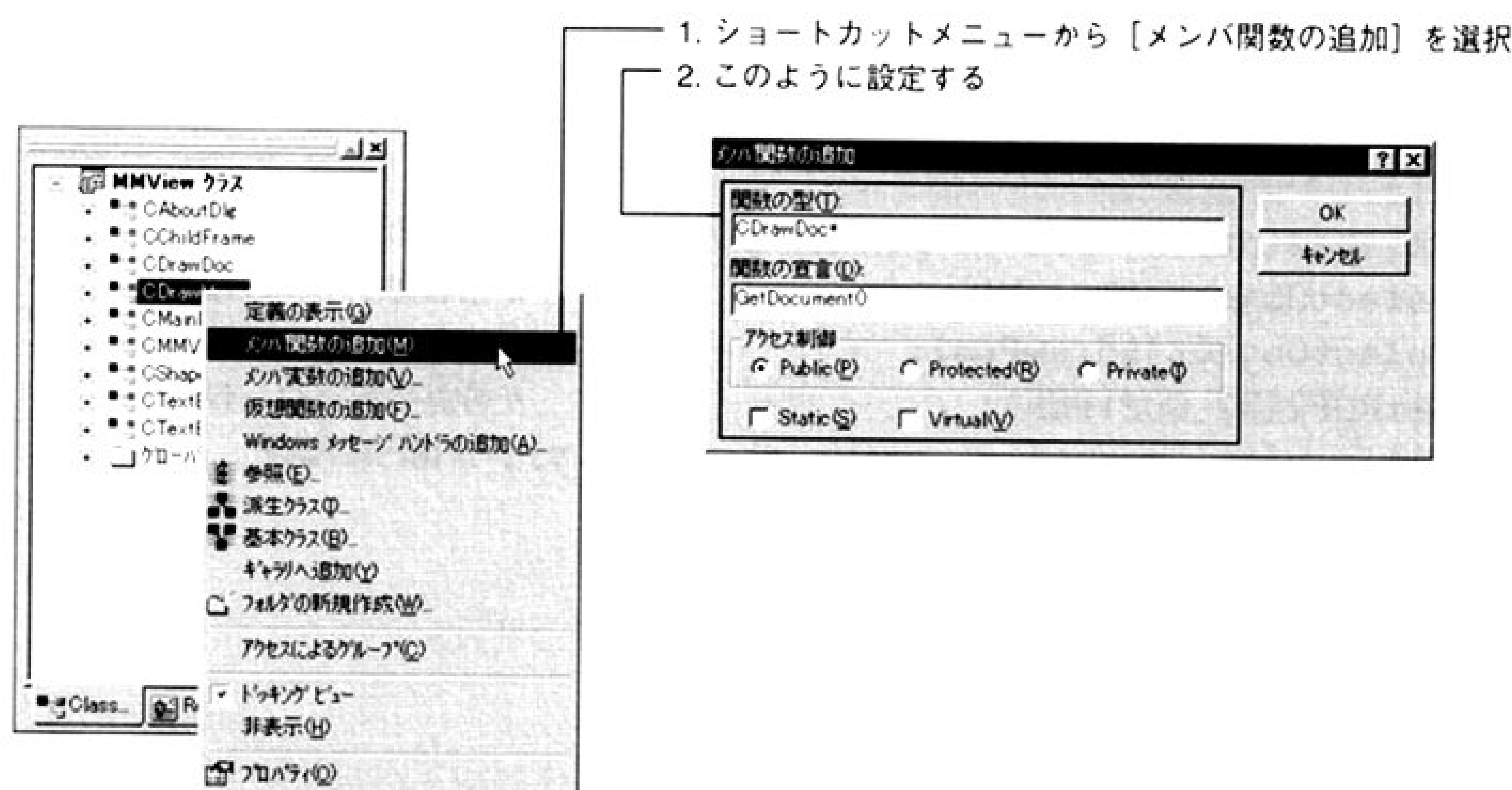


図2-11 ［メンバ関数の追加］ダイアログボックス

リスト2-7 CDrawView::GetDocument関数の実装（DrawView.cpp）

```
CDrawDoc* CDrawView::GetDocument()
{
  return (CDrawDoc*)m_pDocument;
}
```

なお、DrawView.cppの先頭でDrawDoc.hをインクルードするのを忘れないでください（**リスト2-8**）。

リスト 2-8　DrawDoc.h のインクルード(DrawView.cpp)

```
// DrawView.cpp : インプリメンテーション ファイル
//

#include "stdafx.h"
#include "MMView.h"
#include "DrawDoc.h"   // DrawDoc.h をインクルード
#include "DrawView.h"

...
```

リスト 2-7 を見ると、GetDocument 関数は CDrawView クラスのメンバ変数 m_pDocument を返すだけという非常に単純な関数です。なぜそれだけのために関数が必要なのでしょう? なぜかといえば、m_pDocument の型が CDocument クラスへのポインタ(CDocument*型)だからです。たとえば、CDrawView クラスで GetDocument 関数を使うと、CDrawDoc クラスへのポインタが手に入ります。しかし、そうはせずに直接 m_pDocument を参照するなら、参照のたびに m_pDocument をいちいち CDrawDoc*型にキャストしなければなりません。派生クラスのオブジェクトを基底クラスのオブジェクトに代入するときにはキャストは必要ありませんが、逆の場合はかならずキャストが必要だからです。この面倒な作業をせずに済むように、派生クラスへのポインタにあらかじめキャストした値を返す GetDocument 関数が使われるのです。

●コードを記述する位置

最後に、「// アトリビュート」や「// オペレーション」といった ClassWizard が生成したコメントについて説明しておきましょう。これらはしょせんコメントですから、無視して適当な位置にメンバを追加してもコードに影響は与えませんが、MFC のソースコードはこのコメントに応じてメンバの位置を決めているので、プログラマが作成したクラスでも従っておくとよいでしょう。もっとも、実際にはこれ以外に分類されるメンバがいくらでも必要になるはずですから、プログラマが適切なコメントを付ければ、それが読みやすいソースリストにつながります。**表 2-7** に　AppWizard や ClassWizard が生成するコメントとその目的を示しておきます。

コメント	主な利用目的
コンストラクタ	コンストラクタ
アトリビュート	メンバ変数や、メンバ変数への代入／参照に利用する関数
オペレーション	メンバ変数を使って何らかの操作を行う関数
インプリメンテーション	基底クラスの仮想関数をオーバーライドした関数

表 2-7　コメントの意味

2.6 ドキュメントテンプレートの登録

これでリソース／ドキュメントクラス／ビュークラスというドキュメントタイプを構成する部品の用意が終わりました（フレームウィンドウはMFCが提供するCMDIChildWndクラスを利用する）。あとはこれらをまとめてドキュメントテンプレートをテンプレートリストに登録するだけです。これは何も難しいことはありません。以下の行をCMMViewApp::InitInstance関数に付け加えるだけです（**リスト2-9**）。

リスト2-9　ドキュメントテンプレートの登録（MMView.cpp）

```
BOOL CMmviewApp::InitInstance()
{
  ...略
    CMultiDocTemplate* pDocTemplate;
    pDocTemplate = new CMultiDocTemplate(
        IDR_TEXTEDITTYPE,
        RUNTIME_CLASS(CTextEditDoc),
        RUNTIME_CLASS(CChildFrame), // カスタム MDI 子フレーム
        RUNTIME_CLASS(CEditView));
    AddDocTemplate(pDocTemplate);

    pDocTemplate = new CMultiDocTemplate(
        IDR_DRAWTYPE,
        RUNTIME_CLASS(CDrawDoc),
        RUNTIME_CLASS(CMDIChildWnd),
        RUNTIME_CLASS(CDrawView));
    AddDocTemplate(pDocTemplate);

  // メイン MDI フレーム ウィンドウを作成します。
  CMainFrame* pMainFrame = new CMainFrame;
  ...略
}
```

最後に、**リスト2-10**のようにMMView.cppにDrawDoc.hとDrawView.hをインクルードします。これで、とりあえずプログラムが動くようになりました。さっそくコンパイル／実行してみましょう（**図2-12**）。

リスト2-10 2つのファイルのインクルード(MMView.cpp)

```
// MMView.cpp : アプリケーション用クラスの機能定義を行います。
//

#include "stdafx.h"
#include "MMView.h"

#include "MainFrm.h"
#include "TextEditDoc.h"
#include "TextEditView.h"
#include "DrawDoc.h"  // DrawDoc.hのインクルード
#include "DrawView.h" // DrawView.hのインクルード
...
```

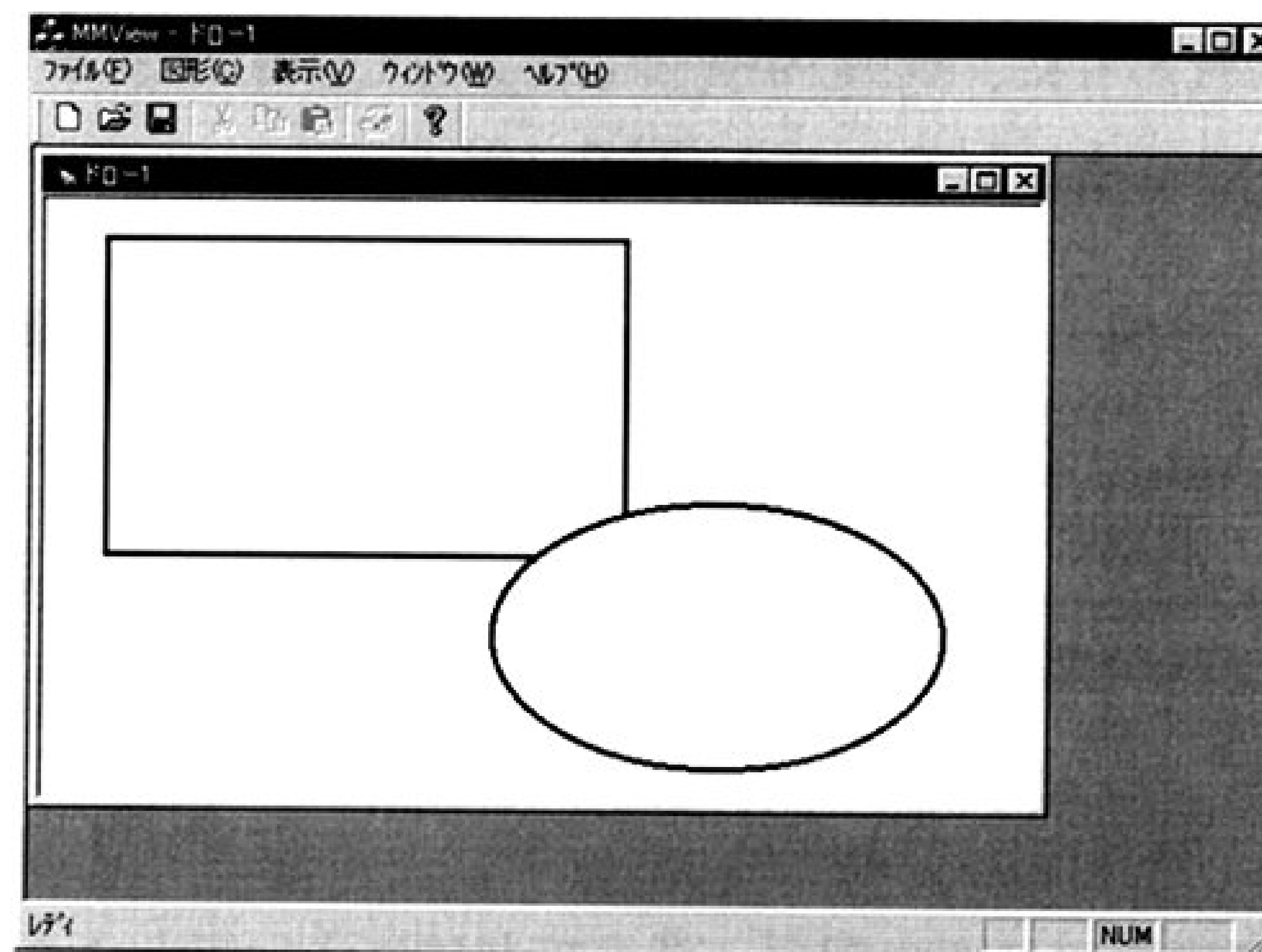


図2-12 実行画面

2.7 描いた図形の保存

これでようやく画面に図形を描くことができるようになりました。とはいえ、今のところウィンドウを最小化すると描いた図形が消えてしまったり、データをファイルに保存できなかったりと、ちょっと寂しいところがあります。

これは画面描画をOnMouseMove関数とOnLButtonUp関数だけで行い、ウィンドウの再描画のことを考えていないところに問題があります。ウィンドウの再描画をする際にはOnDraw関数がかならず呼ばれることを考えると、描いた図形を再描画するコードをOnDraw関数に記述する必要があるでしょう。また、直前に描いた図形以外の図形データを保存していないことも問題です。このままでは、仮に再描画するコードを記述してもたっ

た1つの図形しか再描画できません。本節では、ドキュメントクラスを拡張して、この辺りを改善することにしましょう。そのために、描いた図形のデータをすべて保存することにします。また、ファイルの入出力を行うために、ドキュメントクラスにSerialize関数を実装します。そして最後に、本節でもっとも重要なトピックである、ドキュメントクラス以外のクラスにSerialize関数を実装する方法について解説します。

● リストを使って図形データを保存する

まずは、あとで再描画したりファイルに保存するために、図形データをすべて取っておくことから始めましょう。今のところ、ドキュメントクラスには1つだけしかCShapeクラスのオブジェクトを保存していないので、複数のデータを保存できるように変更する必要があります。そのためには配列を使うのがもっとも簡単ですが、それでは保存できる図形の数に制限ができてしまいます。これはつまらない制限ですから、避けたいところです。そこでここでは配列の代わりに、MFCが提供しているCObListクラスを使って、リスト構造でデータを管理することにしましょう(**図2-13**)。

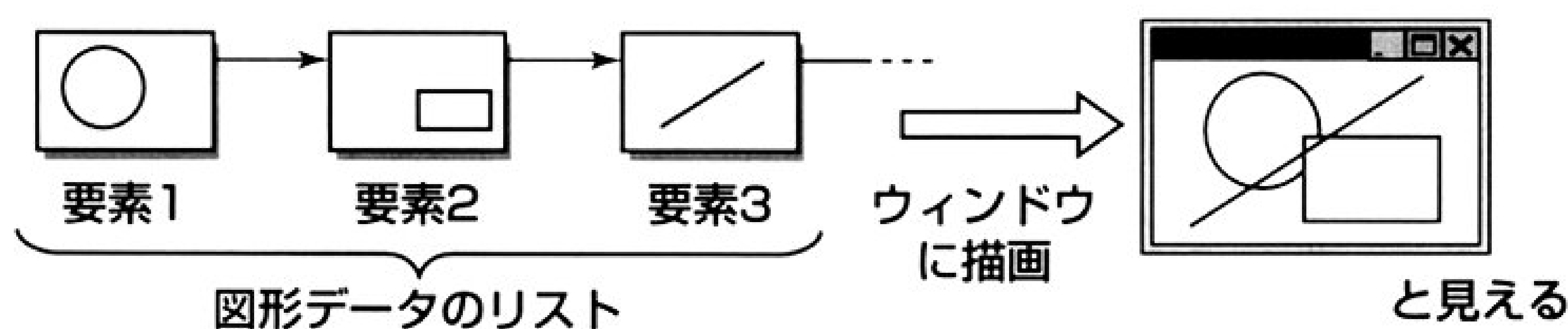


図2-13 リストで図形データを管理する

リスト構造とは、要素数が可変の配列と考えておけばよいでしょう。リストを使うと、CShapeクラスのオブジェクトを必要な数だけ管理できるので、配列のように必要以上に要素をあらかじめ確保しておいたり、要素が足りなくなったらそれまでということもありません。MFCが提供するCObListクラスを使えば、このリスト構造を簡単に扱うことができます。CObListクラスのメンバ関数のうち、本プログラムで利用するものを**表2-8**にあげておきます。

メンバ関数	機能
GetHeadPosition	先頭の要素の位置を返す
GetNext	次の要素の位置を返す
AddTail	リストの末尾に要素を追加する
RemoveAll	すべての要素を削除する
GetCount	要素の数を返す

表2-8 CObListクラスのメンバ関数

それではCObListクラスを使って、入力した図形データをすべて保存できるように変更しましょう。この変更はCDrawDocクラスとCDrawViewクラスの両方に影響を及ぼします。まず図形データを保存するため、CDrawDocクラスに1つのメンバ変数と2つのメンバ関数を追加します。ここでも、プロジェクトワークスペースで[CDrawDoc]を右クリックし、[メンバ変数の追加]および[メンバ関数の追加]を選択してメンバ変数とメンバ関数を追加することが可能です(**図2-14**)。CDrawDoc::AddData関数とCDrawDoc::GetDataList関数の実装は**リスト2-11**に示します。



図2-14 メンバの追加

リスト2-11 CDrawDoc::AddData関数の実装(DrawDoc.cpp)

```
void CDrawDoc::AddData()
{
  CShape* pShape = new CShape;   // CShape オブジェクトの新規作成

  pShape->SetStyle(GetStyle()); // pShape に現在の図形データ (m_shape) をコピー
  pShape->SetPenWidth(GetPenWidth());
  pShape->SetPenColor(GetPenColor());
  pShape->SetBrushColor(GetBrushColor());
  pShape->SetPoint1(GetPoint1());
  pShape->SetPoint2(GetPoint2());

  m_alldata.AddTail((CObject*) pShape);    // pShape を m_alldata に追加
  SetModifiedFlag();                       // モディファイフラグのセット
}

CObList* CDrawDoc::GetDataList()
{
  return &m_alldata;
}
```

図形データを保存するために、CObList クラスのオブジェクトの m_alldata というメンバ変数を用意します。ところで、CObList クラスが管理するリストの各要素の型は CObject クラスですが、ここで扱いたいのは CShape クラスのオブジェクトです。そこで、リストに新しく要素を追加したり、リストの要素を参照したりするときには、CObject クラスと CShape クラスの間でキャストしながら利用することにします。今まで使っていたメンバ変数、m_shape はマウスの左ボタンが離されて 2 点を決定するまでの一時的な作業用オブジェクトとして利用します。

CDrawDoc::AddData 関数は WM_LBUTTONUP メッセージのメッセージハンドラ、CDrawView::OnLButtonUp 関数から呼び出され、図形データのリストの末尾に今描画した図形データを追加するものです。この中で、最後に SetModifiedFlag 関数を呼び出していますが、この関数を実行するとドキュメントの内容が変更されたことがフレームワークに伝えられます。すると、このドキュメントを保存せずに閉じようとすると、フレームワークがメッセージボックスを表示して、ドキュメントを保存するかどうかたずねてくるようになります。

次に CDrawDoc クラスのデストラクタで、オブジェクトが削除されるときにリストに追加された図形データをすべて削除するように変更します（**リスト 2-12**）*1。

リスト 2-12　CDrawDoc クラスのデストラクタの実装（DrawDoc.cpp）

```
CDrawDoc::~CDrawDoc()
{
  POSITION  pos = m_alldata.GetHeadPosition();  // リストの先頭を取得

  while (pos) {
    delete (CShape*)(m_alldata.GetNext(pos));
                   // GetNext 関数の返り値が指すオブジェクトを削除
                   // pos は次の要素を指す
  }
  m_alldata.RemoveAll();
}
```

デストラクタ中では、CObList::GetHeadPosition 関数で先頭の要素の位置を取得し、while ループの中で CObList::GetNext 関数を繰り返し呼び出して、すべての要素を削除します。これとよく似た操作が以前出てきたことを覚えているでしょうか? ドキュメントオブジェクトからビューオブジェクトを取得するために利用した CDocument::GetFirstViewPosition 関数と CDocument::GetNextView 関数がそれです。このような不特定個のデータすべてにアクセスする手法のことを**イテレーション**と呼びます。

*1　デストラクタに処理を記述せずに、CDocument::DeleteContents 関数をオーバーライドすることも考えられるが、ここではデストラクタにデータを削除するためのコードを実装した。

●ビュークラスの変更

次にCDrawViewクラスの変更を行います。CDrawViewクラスに対する変更点は、マウスの左ボタンが離されたらそのときの図形データをリストに追加するように、OnLButtonUp関数内からCDrawDoc::AddData関数を呼び出すようにすることと、リストに登録されているすべての図形データを描画するようにOnDraw関数を変更することの2点です。**リスト2-13**は変更を加えたOnLButtonUp関数です。

リスト2-13　CDrawView::OnLButtonUp関数へのコードの追加(DrawView.cpp)

```
void CDrawView::OnLButtonUp(UINT nFlags, CPoint point)
{
  CDrawDoc* pDoc = GetDocument();
  CDC*  pDC;

  ...中略

  case Elli:
    pDC->Ellipse(pt1.x, pt1.y, pt2.x, pt2.y);
    pDC->SelectObject(&Brush);
    pDC->SelectObject(&drawPen);
    pDC->SetROP2(R2_COPYPEN);
    pDC->Ellipse(pt1.x, pt1.y, point.x, point.y);
    break;
  }
  pDoc->AddData();              //図形データをリストに追加
  pDoc->UpdateAllViews(this); //すべてのビューを更新

  pDC->SelectObject(pOldPen);
  ReleaseDC(pDC);
  ReleaseCapture();
}
```

OnLButtonUp関数内では、CDrawDoc::AddData関数を呼び出した直後にCDocument::UpdateAllViews関数を呼び出しています。これは、1つのドキュメントに対して複数のビューが用意されている場合（[ウィンドウ]－[新規ウィンドウ]などを選択した場合）に必要な処理です。あるビューに対してユーザーが何らかの操作を施すと（この場合はマウスを使って図形を描く）、それによってドキュメントの内容が変更されるわけですが、そのときにはドキュメントの内容を他のビューに反映させる必要があるでしょう。そこで実行するのがCDocument::UpdateAllViews関数です。この関数を呼び出すと、引数に指定したビュー以外のビューのクライアント領域の更新が行われます（**図2-15**）。この場合は、OnLButtonUp関数内ですでにクライアント領域に図形を描画してあるビューに関しては更新する必要がないので、引数にthisを指定しているのです。引数にNULLを指定した場合にはすべてのビューについてクライアント領域の更新が行われます。

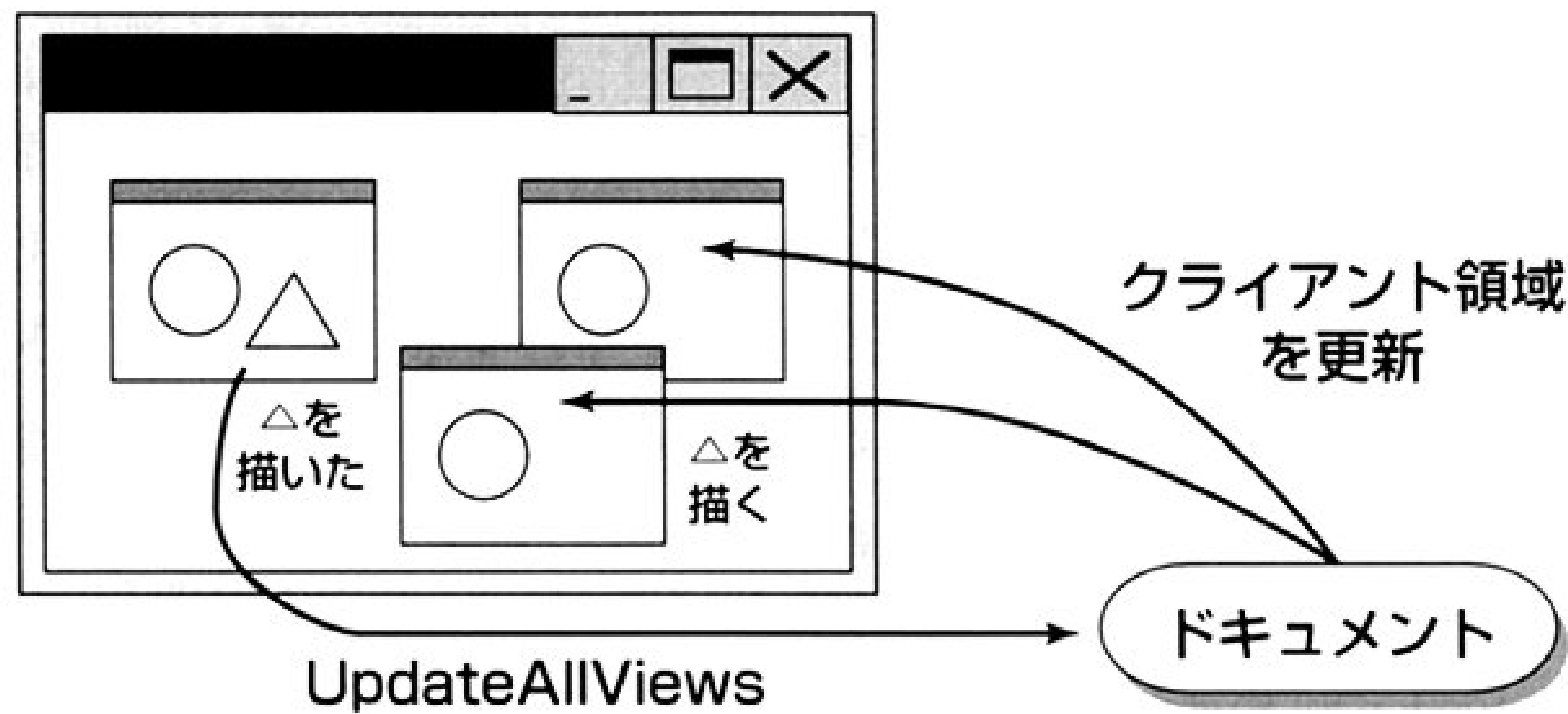


図 2-15　画面更新のメカニズム

CDrawView::OnDraw 関数では、CDrawDoc クラスのデストラクタと同じように、イテレーションを利用して、リストのすべての要素を取得して図形を描画します。そのほかの部分については OnLButtonUp 関数と同様な処理をしています。OnDraw 関数のコードを**リスト 2-14** に示します。

リスト 2-14　変更した CDrawView::OnDraw 関数（DrawView.cpp）

```
void CDrawView::OnDraw(CDC* pDC)
{
  CDrawDoc* pDoc = GetDocument();
  CObList*  pDataList = pDoc->GetDataList();     // 図形データのリストを取得
  POSITION  pos = pDataList->GetHeadPosition(); // リストの先頭を取得
  CShape*   pShape;
  CPen     Pen;
  CPen*    pOldPen;
  CBrush    Brush;
  CBrush*   pOldBrush;
  CPoint    pt1, pt2;

  while (pos) {   // 図形データがある限りは以下の作業を行う
    pShape = (CShape*)(pDataList->GetNext(pos));  // データの取得
    pt1 = pShape->GetPoint1();
    pt2 = pShape->GetPoint2();

    Pen.CreatePen(PS_INSIDEFRAME,pShape->GetPenWidth(),pShape->GetPenColor());
    pOldPen = pDC->SelectObject(&Pen);
    Brush.CreateSolidBrush(pShape->GetBrushColor());
    pOldBrush = pDC->SelectObject(&Brush);

    switch (pShape->GetStyle()) { // 図形の描画
    case Line:
      pDC->MoveTo(pt1);
      pDC->LineTo(pt2);
      break;
```

```
      case Rect:
        pDC->Rectangle(pt1.x, pt1.y, pt2.x, pt2.y);
        break;
      case Elli:
        pDC->Ellipse(pt1.x, pt1.y, pt2.x, pt2.y);
        break;
      }

      pDC->SelectObject(pOldBrush);
      Brush.DeleteObject();
      pDC->SelectObject(pOldPen);
      Pen.DeleteObject();
    }
  }
```

以上で、入力したすべての図形データを保存できるようになったので、もうウィンドウを再描画しても、今までに描いた図形が消えてしまうこともなくなりました。次はCDrawDocクラスにシリアライズを実装して、図形データをファイルとの間でやり取りできるようにします。ここからはドキュメントクラスの中だけの話になってくるので、CDrawView クラスに対する変更はこれで終わりです。残りはすべて CDrawDoc クラスと CShape クラスに対する変更だけになります。

● CDrawDocクラスの Serialize 関数

それでは、前章で扱ったテキストエディタと同じように、シリアライズを実装しましょう。前章で、MFC を使ったプログラムでは、ファイルの入出力はすべてドキュメントクラスの Serialize 関数で行うことを解説しました。そこで、CDrawDoc クラスにも Serialize 関数を実装することにします。

AppWizard が生成したドキュメントクラスには Serialize 関数があらかじめ用意されていましたが、CDrawDoc クラスのように ClassWizard を使って作ったクラスにも Serialize 関数が用意されています。そして、その本体にはやはり次のような CArchive::IsStoring 関数を使ったスケルトンが挿入されています。プログラマはスケルトンとして用意されている if 文の中に書き出し用の処理と読み込み用の処理を追加するだけでよいのは前章で説明したとおりです。

```
void CDrawDoc::Serialize(CArchive& ar)
{
  if (ar.IsStoring())
  {
    // 書き出し用の処理を行う
  }
```

```
    else
    {
        //読み込み用の処理を行う
    }
}
```

コードを記述する前に、まずデータファイルのフォーマットを決めてしまいましょう。保存する必要があるデータは、CDrawDoc クラスのメンバ変数、m_alldata が管理している CShape オブジェクトのリストだけですから、このリストの内容を先頭から順に書き出したり、読み込んだりすればよいことは明らかです。書き出しは確かにそれだけで済みますが、読み込むときには、データファイルにいくつの CShape オブジェクトが保存されているのかわからないと面倒です。そこでデータファイルの先頭には、データファイルに含まれる CShape オブジェクトの数を保存することにしましょう。

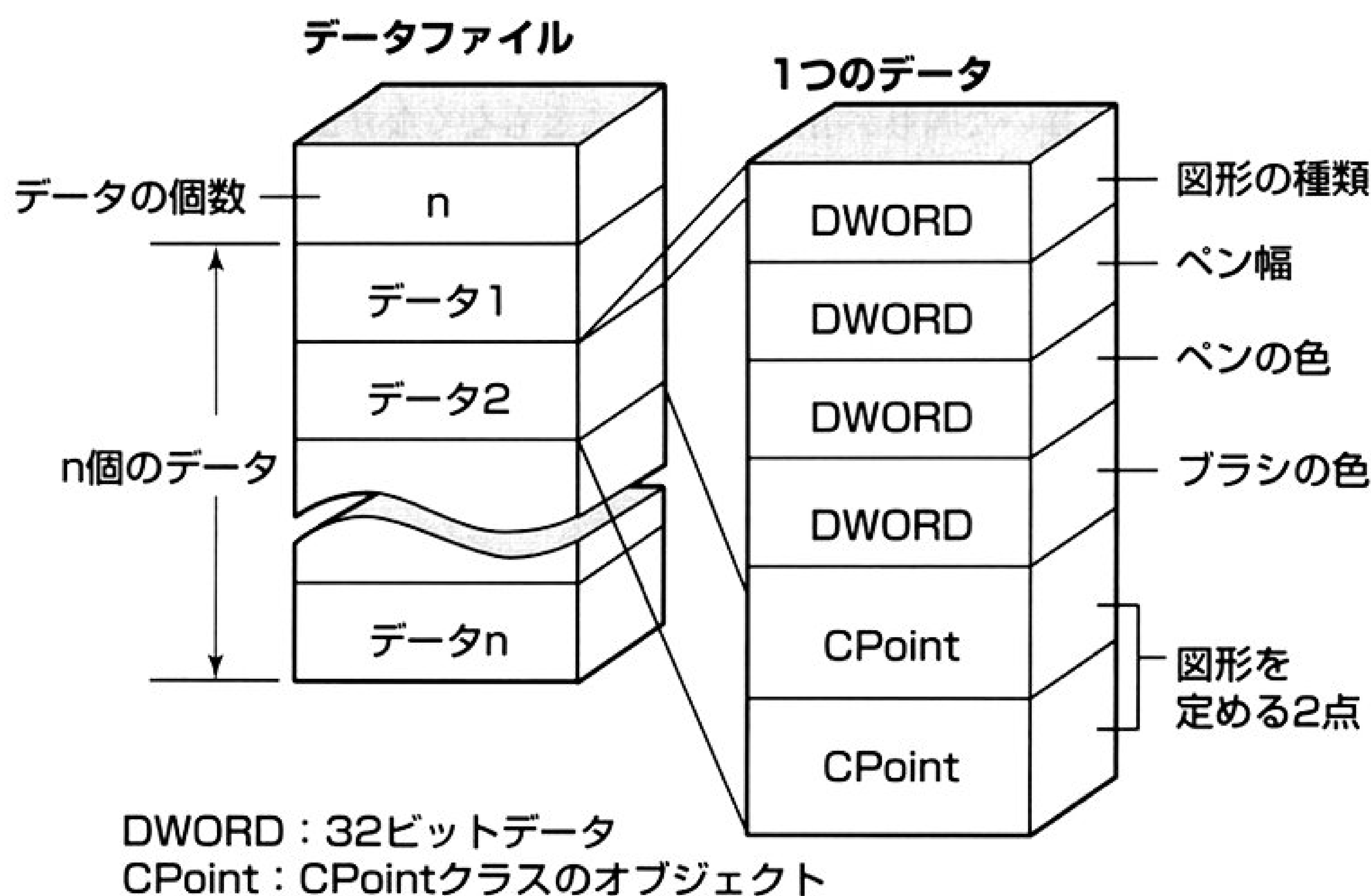


図2-16 データファイルのフォーマット

データの書き出し

データフォーマットが決まったので、まず書き出し用の処理から記述します。リストの要素数は CObList::GetCount 関数で調べることができますから、この値を ar オブジェクトに書き出します。次にリストの各要素を書き出すわけですが、ここでも CDrawView::OnDraw 関数で行ったように、CObList::GetHeadPosition 関数と CObList::GetNext 関数を使ってイテレーションを行います。そして、手に入れた CShape オブジェクトへのポインタを使って、CShape オブジェクトのメンバ変数をすべて書き出します。

CDrawDocクラスで追加した4つのメンバ変数（m_style、m_penwidth、m_pencolor、m_brushcolor）はすべて本質的にはDWORD型（32ビットの符号なし整数）と同等ですから、DWORD型にキャストすれば、CArchiveクラスのメンバ関数として定義されている<<演算子（引数の型がDWORDのもの）を使ってファイルに書き出すことができます。しかし、m_point1メンバ変数とm_point2メンバ変数の2つはCPoint型のオブジェクトです。CPoint型のオブジェクトのための<<演算子はCArchiveクラスには用意されていません。にもかかわらずCDrawDoc::Serialize関数では、平気な顔をして<<演算子を使っています。これは別にC++言語の未知の機能を使っているわけではなく、単に別の場所（Microsoft Visual Studio¥VC98¥Mfc¥include¥Afxwin1.inl）で定義されているだけです。また、CArchiveクラスのメンバ関数としてではなく、グローバル関数として定義されているため、マニュアルを引いても出てこないのです。

MFCのソースコードを読むときにどうしても見つからない関数があったら、「Microsoft Visual Studio¥VC98¥Mfc¥src」フォルダにある.cppファイルだけではなく、「Microsoft Visual Studio¥VC98¥Mfc¥include」内の.inlファイル（インライン関数が定義されている）を疑ってみるとよいでしょう。

データの読み込み

さて、次に読み込み用の処理を記述します。こちらは、

1. CShapeオブジェクトをnew演算子を使って用意する
2. arオブジェクトを使って、各メンバ変数の値を読み込む
3. CObList::AddTail関数を呼び出して、ドキュメントが管理するリストにCShapeオブジェクトを追加する

という処理をファイルに保存されているCShapeオブジェクトの数だけ繰り返します。以上の処理をまとめたものが**リスト2-15**に示すコードです。

リスト2-15　CDrawDoc::Serialize関数の実装（DrawDoc.cpp）

```
void CDrawDoc::Serialize(CArchive& ar)
{
  CObList*  pDataList = GetDataList();  //図形データのリストを取得
  CShape*   pShape;

  if (ar.IsStoring())
  {
    ar << (DWORD) pDataList->GetCount();  //要素数を出力
    POSITION pos = pDataList->GetHeadPosition();
    while (pos) {
      pShape = (CShape*) pDataList->GetNext(pos);
```

```
        ar << (DWORD)pShape->m_style;
        ar << (DWORD)pShape->m_penwidth;
        ar << (DWORD)pShape->m_pencolor;
        ar << (DWORD)pShape->m_brushcolor;
        ar << pShape->m_point1;
        ar << pShape->m_point2;
      }
    }
    else
    {
      int count;
      ar >> (DWORD&) count;
      while (--count >= 0) {
        pShape = new CShape;
        ar >> (DWORD&)pShape->m_style;
        ar >> (DWORD&)pShape->m_penwidth;
        ar >> (DWORD&)pShape->m_pencolor;
        ar >> (DWORD&)pShape->m_brushcolor;
        ar >> pShape->m_point1;
        ar >> pShape->m_point2;
        pDataList->AddTail((CObject*) pShape);
      }
    }
  }
```

このとき、注意して欲しいのはCShapeクラスの6つのメンバ変数はすべてプライベートメンバだったということです。通常なら、CDrawDocクラスのメンバ関数からはSet～関数かGet～関数を介してしか、これらのメンバにはアクセスできないはずなのにここでは何の問題もなくアクセスできています。これがfriendキーワードの効果です。CDrawDocクラスがCShapeクラスのフレンドクラスでないときには、シリアライズ処理はもっと面倒なものになっているはずです。

もっとも、これはクラスの設計にミスがあったというべきかもしれません。CShapeクラスのようなクラスはむしろ単なる構造体か、もしくはすべてがパブリックメンバから構成されるクラスで管理をするべきでしょう。シリアライズの実装が終了した、ここまでのプログラムは「MMView¥Step2」フォルダに収めてあります。

● CShapeクラスにSerialize関数を実装する

さて、図形の再描画やファイルとの入出力もできるようになって、これでドローツールの完成！といいたいところですが、あと1つ解説したいことがあるので、もう少しおつきあいください。それは、「CShapeクラスにSerialize関数を実装する方法」についてです。

「えっ？ Serialize関数ってCDocumentクラスのメンバ関数なんじゃないの？」と思っ

た方は勘違いをしています。Serialize関数はCDocumentクラスのメンバ関数ではなく、CObjectクラスのメンバ関数です。CObjectクラスとは、MFCが提供するほとんどすべてのクラスの基底クラスとなっている、ルートクラスとも呼べるクラスです。つまり、MFCが提供するほとんどのクラスにはSerialize関数が潜在的に存在しているのです。ただ、必要のないクラスには実装されていないために、表面上には現れないだけなのです。

シリアライズ可能クラスのメリット

CShapeクラスにSerialize関数を実装すると、どういったメリットがあるのでしょう? くどくど説明するよりも、CShapeクラスにSerialize関数を実装した場合に、CDrawDoc::Serialize関数がどのように変わるか、見ていただきましょう(**リスト2-16**)。

リスト2-16 変更したCDrawDoc::Serialize関数(DrawDoc.cpp)

```
void CDrawDoc::Serialize(CArchive& ar)
{
  CObList*  pDataList = GetDataList();
  CShape*   pShape;

  if (ar.IsStoring())
  {
    ar << (DWORD) pDataList->GetCount();
    POSITION pos = pDataList->GetHeadPosition();
    while (pos) {
      pShape = (CShape*) pDataList->GetNext(pos);
      ar << pShape;   // ドキュメントクラスでのシリアライズはこれだけ
    }
  }
  else
  {
    int count;
    ar >> (DWORD&) count;
    while (--count >= 0) {
      ar >> pShape;   // ドキュメントクラスでのシリアライズはこれだけ
      pDataList->AddTail((CObject*) pShape);
    }
  }
}
```

今まではCShapeクラスのメンバ変数を1つ1つarオブジェクトを使って読み書きしていたのが、CShapeオブジェクトがまさに1つのオブジェクトとしてarオブジェクトを使って読み書きできるようになっているのがわかります。これだけでも、CShapeオブジェクトを読み書きしていることがはっきりして、非常にプログラムがわかりやすくなっています。本プログラムのように比較的単純なデータ構造を管理するドキュメントクラスでさえ、これだけの効果があるのですから、より複雑なデータ構造を扱うドキュメントクラス

ならば、見通しのよさが格段に違ってきます。

また、ドキュメントクラス以外のクラスにSerialize関数を実装した場合には、そのクラスのオブジェクトを書き出すときには、フレームワークによってデータの前にクラスの識別子が出力されます。この識別子は読み込み時に参照され、フレームワークにより型チェックが行われます。このように、安全にファイルの入出力を行うためのサービスも受けることができるようになるのです。

Serialize関数が呼び出されるとき

ここまでの説明を読んで、「なぜCArchiveクラスのオブジェクトを使って読み書きすると、Serialize関数が呼び出されるのだろう？」と疑問に思っていた方も多いかと思います。ここでは、Serialize関数が呼び出されるまでのフレームワーク内の動作について、ドキュメントクラスとそれ以外のクラスに分けて説明します。ドキュメントクラスとそれ以外のクラスでは、Serialize関数の呼び出され方が違うことに注意してください。

まずドキュメントクラスの場合を説明しましょう。ドキュメントクラスのSerialize関数は、フレームワークが用意したファイルオープンダイアログボックスなどを使ってファイルをオープンしたときに呼び出されることはすでにお話ししました。

たとえば、メニューから［ファイル］－［開く］を選択した場合を考えてみましょう。AppWizardが生成したコードでは、このメニュー項目を選択すると、フレームワークによってCWinApp::OnFileOpen関数が呼び出され、ファイルオープンダイアログボックスが表示されます。次に、OnFileOpen関数から、ダイアログボックスでユーザーが指定したファイル名を引数としてCDocument::OnOpenDocument関数が呼び出されます。ここでは、CArchiveオブジェクトの作成、ファイルのオープン、Serialize関数の呼び出し、ファイ

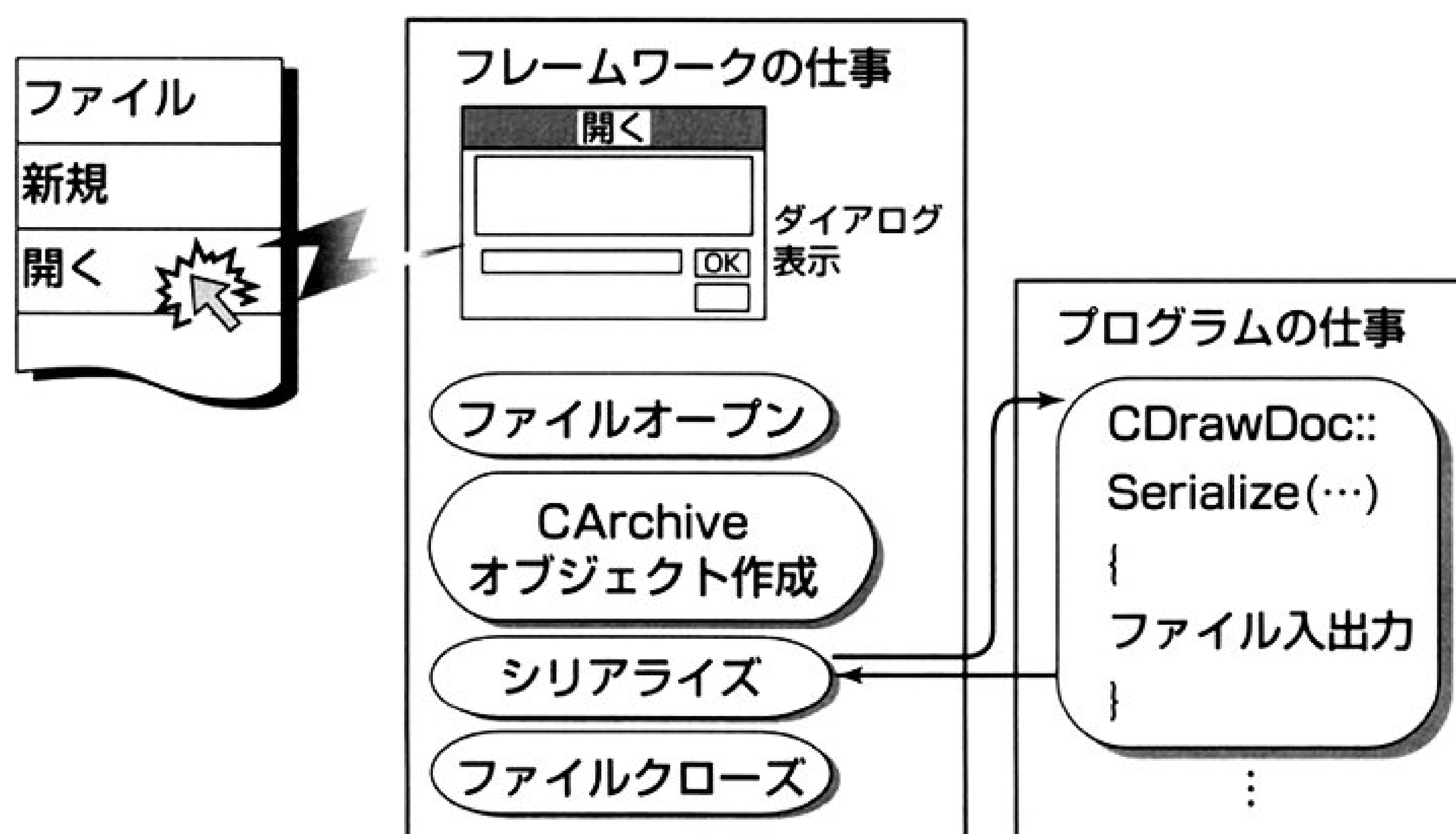


図2-17　ダイアログボックスを通して呼び出される場合

ルのクローズが順に行われます（**図 2-17**）。つまり、ドキュメントクラスの Serialize 関数は CArchive クラスの＜＜演算子や＞＞演算子を使わずに、フレームワークから直接関数名を指定して呼び出されるのです。

以上は読み込みの場合でしたが、［ファイル］－［保存］を選択したときも、フレームワークによって書き出し用に、ほぼ同様の処理が行われます。

次にドキュメントクラス以外のクラスの場合についてです。これらのクラスの Serialize 関数は、CArchive オブジェクトを通して、＜＜演算子や＞＞演算子などを使って読み書きをすると呼び出されます。

例として、CShape クラスに Serialize 関数を実装したものとして説明しましょう。シリアライズが実装されたオブジェクトの入出力には、CArchive オブジェクトを使えることはすでに説明しました。その具体的な使い方は**リスト 2-16** でも紹介していますが、次のようにして使います。

```
CArchive ar;
CShape* pShape;
ar << pShape;    // CShape オブジェクトの書き出し
ar >> pShape;    // CShape オブジェクトの読み込み
```

さて、ここからどのようにして CShape::Serialize 関数が呼び出されるのか、読み込みの場合を例にして追っていくことにします。「ar >> pShape;」のように＞＞演算子の左項に CArchive オブジェクト、右項に CShape オブジェクトへのポインタを配置すると、＞＞演算子によって実際に実行されるのは

```
CArchive& operator>>(CArchive& ar, CShape* pShape);
```

です。しかし、このような関数がフレームワークに用意されているはずがありません。なにしろ CShape クラスは私達が作ったクラスなのですから。といってもこのような関数をプログラマが定義する必要はありません。実は、フレームワークに用意されているマクロ（DECLARE_SERIAL と IMPLEMENT_SERIAL）を使えば、関数のプロトタイプ宣言から関数本体の定義までやってくれるのです。マクロによって定義された関数の本体では、CArchive::ReadObject 関数を呼び出しています。そして、CArchive::ReadObject 関数の中から、CShape::Serialize 関数が呼び出されるのです。

以上が＞＞演算子での読み込みによって Serialize 関数が呼び出されるまでの経路です。次に書き出しの場合について見てみましょう。今度は読み込みの場合とは違って、プログラマが（マクロを使って）＜＜演算子を用意する必要はありません。実際には、フレームワークから次の関数が呼び出されます。

```
CArchive& operator<<(CArchive& ar, const CObject* pOb)
```

そして、第2引数、CObjectクラスのオブジェクトへのポインタが指すオブジェクトのSerialize関数が呼び出されます。よって、Serialize関数を実装するクラスは、CObjectクラスの派生クラスでなければなりません。とはいっても、Serialize関数はそもそもCObjectクラスのメンバ関数ですから、当然といえば当然です。このように、読み込み／書き出しで違いがあるのは、「派生クラスのポインタは基底クラスのポインタに代入できるが、基底クラスのポインタは派生クラスのポインタに代入できない」というC++言語の仕様があるためです。ここでは詳しく説明しませんので、よく考えてみてください。

上記の関数の本体では、CArchive::WriteObject関数を呼び出しています。そして、CArchive::WriteObject関数の中から、CShape::Serialize関数が呼び出されます。以上がドキュメントクラス以外のクラスに実装したSerialize関数が呼び出されるまでの流れでした。

ところで、ドキュメントクラスのSerialize関数はフレームワークから呼び出されますが、ドキュメントクラス以外のクラスのSerialize関数が呼び出されるきっかけとなる、<<演算子と>>演算子はいつ使われるのでしょう？ それは、ほとんどの場合ドキュメントクラスのSerialize関数の中となるはずです。フレームワークからドキュメントクラスのSerialize関数が呼び出されると、そこでプログラマが記述したコードが実行されることになります。プログラマは、おそらくドキュメントクラスのメンバ変数をファイルに保存しようとして、<<演算子と>>演算子を使うでしょう。プログラマがそのメンバ変数のクラス（この例ではCShapeクラス）についてきちんとシリアライズを実装していれば、ここでそのクラスのSerialize関数が呼び出されるわけです*2。

CShapeクラスをシリアライズ可能クラスに書き換える

あるクラスにSerialize関数を実装するために必要な条件を抽出すると以下のようになります。

- そのクラスはCObjectクラスの派生クラスである（直接である必要はない）
- DECLARE_SERIALマクロとIMPLEMENT_SERIALマクロを利用している

これらの条件を満たすように、DrawDoc.hで定義されているCShapeクラスを**リスト2-17**のように変更しましょう。

*2 もしくはドキュメントクラスのSerialize関数の中でarを引数にシリアライズを行うオブジェクトのSerialize関数をそのまま呼び出すことも可能。

リスト2-17 変更したCShapeクラスの定義(DrawDoc.h)

```
class CShape : public CObject //CShapeクラスをCObjectクラスの派生クラスとした
{
  friend class CDrawDoc;
  DECLARE_SERIAL(CShape)    //シリアライズ可能クラスであることを宣言する

private:
  STYLE    m_style;         //図形の種類
  int      m_penwidth;      //ペンの太さ
  COLORREF  m_pencolor;     //ペンの色
  COLORREF  m_brushcolor;   //ブラシの色
  CPoint    m_point1;       //第1の点の座標
  CPoint    m_point2;       //第2の点の座標

public:
  void    SetStyle(STYLE s)          { m_style = s; }
  void    SetPenWidth(int w)         { m_penwidth = w; }
  void    SetPenColor(COLORREF c)    { m_pencolor = c; }
  void    SetBrushColor(COLORREF c)  { m_brushcolor = c; }
  void    SetPoint1(CPoint pt)       { m_point1 = pt; }
  void    SetPoint2(CPoint pt)       { m_point2 = pt; }

  STYLE     GetStyle()       { return m_style; }
  int       GetPenWidth()    { return m_penwidth; }
  COLORREF  GetPenColor()    { return m_pencolor; }
  COLORREF  GetBrushColor()  { return m_brushcolor; }
  CPoint    GetPoint1()      { return m_point1; }
  CPoint    GetPoint2()      { return m_point2; }

  void Serialize(CArchive& ar);    //Serialize関数を宣言する
};
```

ポイントは、

- CShapeクラスをCObjectクラスの派生クラスに変更
- 「DECLARE_SERIAL(CShape)」という1行を追加
- Serialize関数を追加

の3点です。DECLARE_SERIALマクロの最後にはセミコロン(;)は必要ないことに注意してください。あとはDrawDoc.cppにCShape::Serialize関数を実装し、マクロを1つ追加するだけです。この場合のSerialize関数も今までと同じような、標準的な構造をしています(**リスト2-18**)。

リスト2-18　CShape::Serialize関数の実装(DrawDoc.cpp)

```
/////////////////////////////////////////
// CShape シリアライズ

IMPLEMENT_SERIAL(CShape, CObject, 0)

void CShape::Serialize(CArchive& ar)
{
  if (ar.IsStoring())
  {
    ar << (DWORD) m_style;
    ar << (DWORD) m_penwidth;
    ar << (DWORD) m_pencolor;
    ar << (DWORD) m_brushcolor;
    ar << m_point1;
    ar << m_point2;
  }
  else
  {
    ar >> (DWORD&) m_style;
    ar >> (DWORD&) m_penwidth;
    ar >> (DWORD&) m_pencolor;
    ar >> (DWORD&) m_brushcolor;
    ar >> m_point1;
    ar >> m_point2;
  }
}
```

CShape::Serialize関数の内容は、以前のCDrawDoc::Serializeメンバ関数で行っていた処理をそのまま持ってきただけであることがわかると思います。以上で、CShapeクラスへのシリアライズの実装は終わりです。ここまでのプログラムは「MMView¥Step3」ディレクトリに収めてあります。

第4部 Windowsらしいアプリケーションを作ってみよう

第3部に引き続き、第4部でもちょっとしたアプリケーションを1つ作成します。第3部で作成したMMViewのキーワードは「ドキュメント－ビュー・アーキテクチャ」でしたが、今回のキーワードは「コントロール」です。コントロールといえば、Windowsアプリケーションのユーザーインターフェイスを担う、いわば、アプリケーションの顔です。Windows 95以降、少しずつWindowsアプリケーションのデザインが変わってきているように感じられるのは、ちゃくちゃくと追加される新規コントロールによるところが大きいようです。

コントロールの使い方（DDX／DDV）についてはすでに第2部で解説を終えていますが、第4部ではより突っ込んだコントロールの使い方について解説を行います。DDX／DDVは手軽である反面、できることも限られています。コントロールを使いこなすには、DDX／DDVを使わない方法を学ぶ必要があります。

第4部では、ツリービューコントロールとHTMLコントロールを題材に、細かなコントロールの使い方を解説します。

1 URLマネージャの概要

Visual C++ではDDX/DDVという機構を使うことによって、ダイアログボックスなどでのユーザーインターフェイスをほとんどコントロールごとの違いを意識せずに作ることができます(これについては第2部で解説した)。ただし、これは「コントロールが持つ値を設定、取得する」というごくごく単純な作業だけをする場合に限られます。エディットボックスやリストボックスのようなダイアログボックスで使用する類の簡単なコントロールならば、確かにDDX／DDVは有効な手段です。しかし、現在ではコントロールの機能は複雑化する一方です。たとえば、エクスプローラで使われているツリービューコントロール(ディレクトリツリー(左側のペイン)で使われている)やリストビューコントロール(ファイルリスト(右側のペイン)で使われている)などの場合、「コントロールが持つ値を設定、取得する」だけでは制御しきれないことは一目瞭然です。ざっと考えてみても、アイテムの挿入、ビットマップの設定、ラベルの編集など、とてもDDX／DDVだけでは扱いきれない作業が必要であることがわかります。

そこで、第4部ではDDX／DDVを使わずに、コントロールを制御する方法をツリービューコントロールを題材に解説します。またVisual C++ 6.0で追加されたHTMLコントロールも簡単に使用します。これらのコントロールをDDX／DDVを使わずに制御する場合は、各コントロールに依存する作業が必要になるので、解説もコントロールごとに依存するものになります。したがって、第4部で解説できるコントロールはツリービューコントロールに限られてしまいますが、ご了承ください。

1.1 URLマネージャの使い方

さて、上述したようなコントロールの使い方を解説するために用意したサンプルアプリケーションを紹介しましょう。このアプリケーションは「URLマネージャ」と名付けられ、**図1-1**のような外観をしています。



図 1-1 URL マネージャ

エクスプローラのようにメインウィンドウは 2 つの領域に分けられていて、左側の領域にはツリー状に URL のリストを表示し、右側の領域には Web ページを表示します。この左側の領域に使われているコントロールがツリービューコントロールです。また、右側の領域に使われているコントロールが HTML コントロールです。ちょうど、エクスプローラのエクスプローラバーに「お気に入り」を表示したときと同等の機能を持ったアプリケーションです。

URL マネージャに表示する URL リストは、Internet Explorer 4.0（以後 IE4.0 と略す）で管理されている「お気に入り」から作成しますが、プログラムを単純化するために、URL マネージャ自身には「お気に入り」を取得する機能を持たせていません。そこで、代わりに lsurl.exe というツールを用意しました。lsurl ツールを起動すると、現在ログオンしているユーザーの「お気に入り」を調べて、登録されている URL とタイトル、それにフォルダを抜き出して 1 つのテキストファイルを新規作成して記録します。URL マネージャは、この lsurl ツールが作成したテキストファイルを読み込んで、ツリービューに表示します。lsurl ツールについては本書が扱う範囲を越える内容になるので解説は割愛しますが、ソースコードは付属 CD-ROM に収録されているので（URLMan¥lsURL）、興味があれば解析してみてください。

1.2 URLマネージャの構造

AppWizard を起動してスケルトンを作成する前に、URL マネージャの大まかな構造を解説しておきましょう。

AddressBook のウィンドウは 2 つの領域に分かれていると述べましたが、このような 1 つのウィンドウの中にしきいを用意して複数の領域に分けて使うテクニックをスプリットウィンドウと呼びます（**図 1-2**）。スプリットウィンドウのそれぞれの領域のことはペインと呼びます。ペイン（pane）とは窓ガラスのことです。ウィンドウの中にある区切られた領域だからペイン（窓ガラス）というわけです。

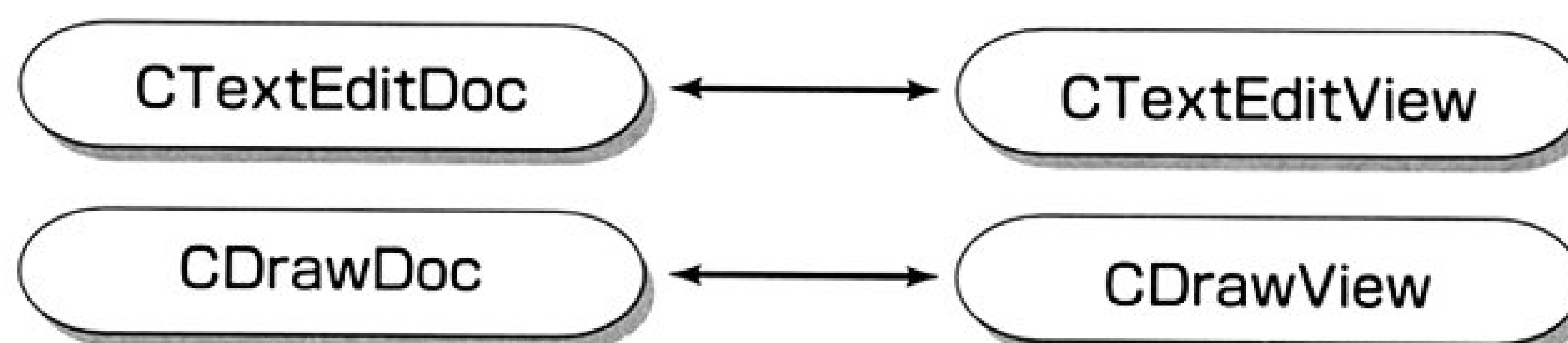


図 1-2　スプリットウィンドウ

こうして作られたペインのそれぞれにはビュークラスを割り当てて、通常のビューウィンドウとまったく同じように扱うことができます。つまり、ペインを複数用意すれば、1 つのウィンドウ（アプリケーション）で複数のビュークラスを利用できるのです。第 3 部の MMView でもビュークラスを複数使っていましたが、このときはドキュメントクラスも複数存在していて、1 つのビュークラスに 1 つのドキュメントクラスが対応していました。しかしスプリットウィンドウを使うときには、1 つのドキュメントクラスに複数のビュークラスが対応することになります（**図 1-3**）。

URL マネージャでは CURLTreeView クラスと CBrowserView クラスという 2 つのビュークラスを作成します。CURLTreeView クラスは CTreeView クラスの派生クラスと

MMViewの場合
ドキュメントクラスとビュークラスが1対1に対応



URLマネージャの場合
1つのドキュメントクラスに2つのビュークラスが一関係している

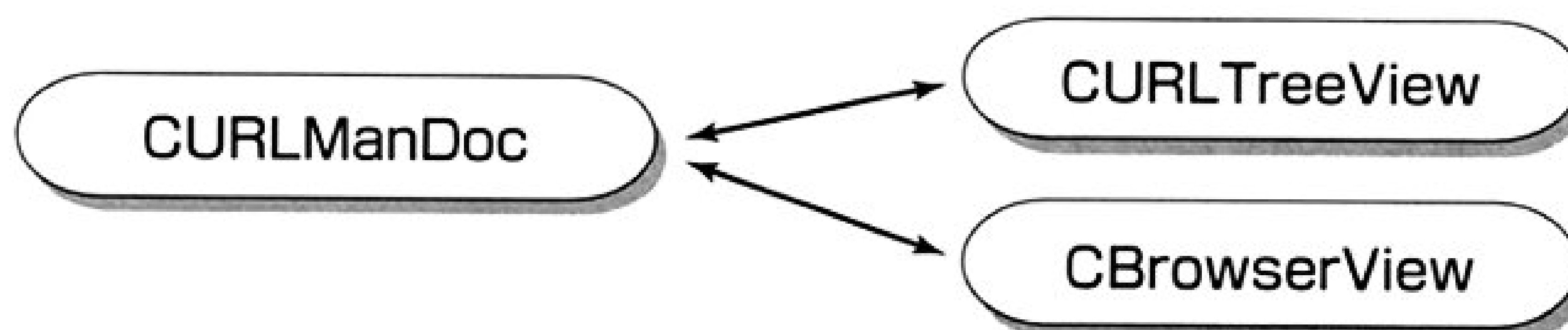


図 1-3　MMView と URL マネージャ

して定義します。CTreeView クラスは、ツリービューコントロールをビュークラスとして使うためのクラスです。また CBrowserView クラスは CHtmlView クラスの派生クラスとして定義します。CHtmlView クラスも CTreeView クラスと同じように、HTML コントロールをビュークラスとして使うためのクラスです。

HTML コントロールの機能は、IE 4.0 とまったく同じです。HTML のサポートタグは完全に一致しますし、ActiveX コントロールや Java もサポートされています。それに、プロキシーサーバーの設定は IE 4.0 のものを拝借します。IE 4.0 も HTML コントロールを使用して作られているので、これらは当然のことです。

これら 2 つのクラスは、第 3 部で使用した CEditView クラスと同じように、1 つのコントロールをまるまるビュークラスとして扱うために用意されたクラスです。ダイアログボックスでコントロールを使うときのようにリソースを作成する必要はなく、ビューオブジェ

クラス名	用途
CURLManApp	アプリケーションクラス
CURLManDoc	ドキュメントクラス
CURLTreeView	URL のツリー表示を行う
CBrowserView	ツリービューで選択されている URL の内容を表示する
CMainFrame	フレームウィンドウクラス
CAboutDlg	アバウトダイアログボックス
CURLEntry	URL を記憶するために使用する

表 1-1　URL マネージャで使用するクラス一覧

クトが作成されると同時にコントロールも作成され、ビューウィンドウ全体を占めるようにコントロールが配置されます。

以上2つのクラスのほかにも、通常のSDIアプリケーションで使用される**表1-1**に示すようなクラス、およびドキュメント用補助クラスが1つ存在します。これらのクラスについては2章以降で解説していきます。

1.3 スケルトンを作る

それではいつものようにAppWizardを使ってスケルトンを作成することにしましょう。[プロジェクト名]は[URLMan]とします。ほとんどの設定項目はデフォルトのままですが、ステップ1の[作成するアプリケーションのタイプ]をSDIに、ステップ6のビュークラスの名前と基底クラスをそれぞれ、CURLTreeViewクラスとCTreeViewクラスに変更します(**図1-4**)。

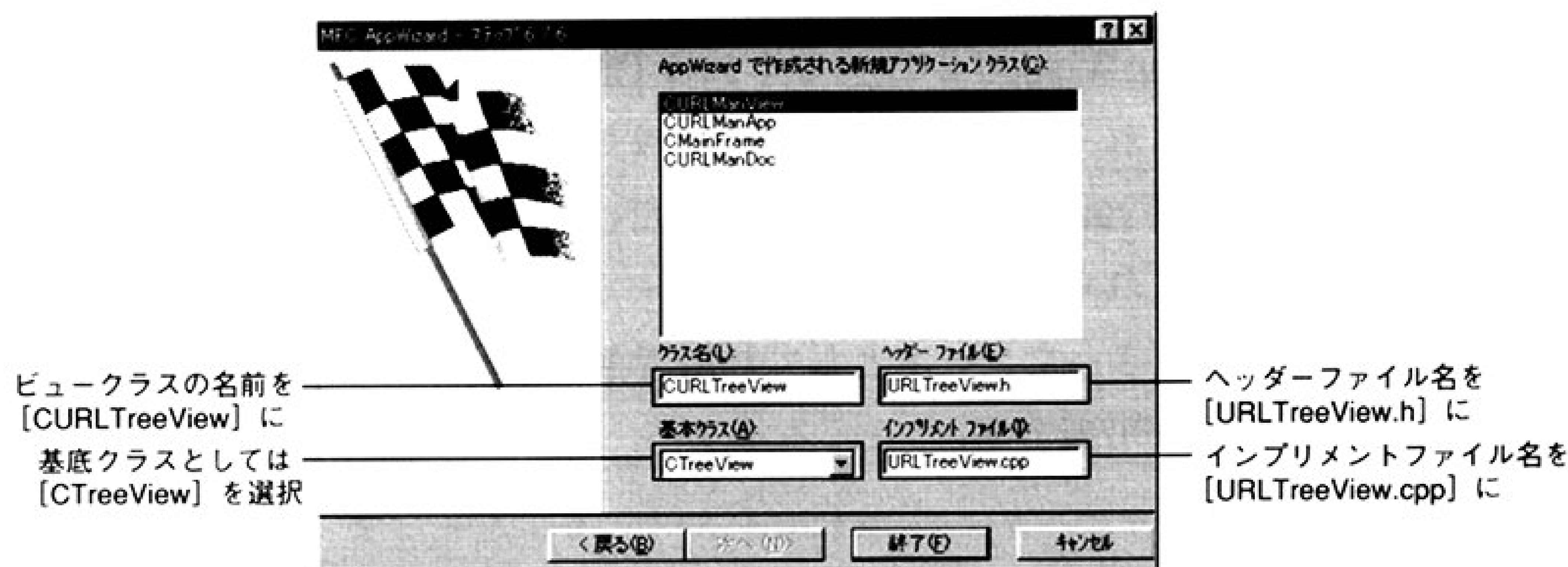


図1-4 AppWizardでの変更箇所

以上の設定を行ったらスケルトンの完成ですが、今回は2つのビュークラスを使用しますから、続けてClassWizardを使ってCBrowserViewクラスを作成することにしましょう。ClassWizardを起動して、<クラスの追加>ボタンをクリックして[新規]を選択すると[新規クラスの作成]ダイアログボックスが開くので、[クラス名]に[CBrowserView]を、[基本クラス]に[CHtmlView]をそれぞれ指定してください。最後に<OK>ボタンをクリックしたらCBrowserViewクラスの完成です。

AppWizardが手助けしてくれるのはここまでです。これから先はプログラマが1つ1つコードを作っていく番です。次に、第4部でのプログラマ側の初仕事として、URLマネージャでスプリットウィンドウを使用できるように加工をしておくことにしましょう。

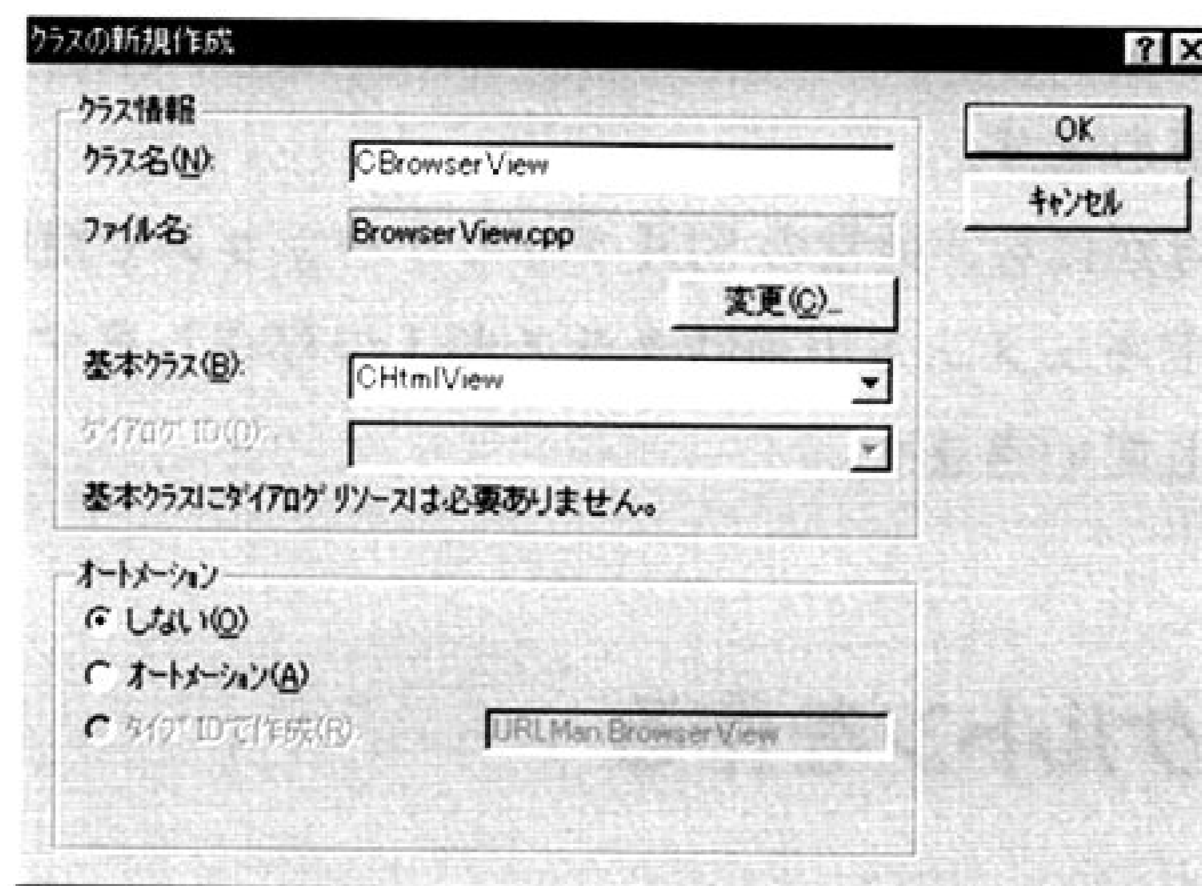


図 1-5　ClassWizard で CBrowserView クラスを作る

1.4 スプリットウィンドウを作成する

スプリットウィンドウはとても簡単に作ることができます。あらかじめ作っておく必要のあるクラスは、通常 AppWizard が作ってくれる CMainFrame クラス、そして 2 個以上のビュークラスです。これらはここまでの作業ですでに用意されています。あとはごくわずかなコードを CMainFrame クラスに追加するだけで、スプリットウィンドウが利用できるようになります。これらのコードはアプリケーションによらず、スプリットウィンドウを使う場合には決まりきったコードとして利用できるので、URL マネージャ以外のプロジェクトにも使い回しが可能です。

● メンバ変数の追加

まずは、スプリットウィンドウを利用するために必要なメンバ変数を追加します。まずワークスペースウィンドウの[ClassView]ページで[CMainFrame クラス]をダブルクリックするなどして、MainFrm.h を開いてください。するとそこには CMainFrame クラスの定義が記述されているので、そこに次の 1 行を追加してメンバ変数を追加してください(**リスト 1-1**)。

リスト 1-1　メンバ変数の追加(MainFrm.h)

```
class CMainFrame : public CFrameWnd
{
...
public:
    CSplitterWnd m_wndSplitter;
}
```

m_wndSplitter メンバ変数は CSplitterWnd クラスのオブジェクトで、フレームウィンドウのクライアント領域を乗っ取る目的で使われます。通常フレームウィンドウのクライアント領域はビューウィンドウが管理しています。これは CMainFrame クラスの基底クラスである CFrameWnd クラスの OnCreateClient メンバ関数の中で、フレームウィンドウのクライアント領域に指定されたビューウィンドウを割り当てる、という処理を行っているからです。CFrameWnd::OnCreateClient メンバ関数はフレームウィンドウのクライアント領域を初期化するための関数ですから、CMainFrame クラスでこのメンバ関数をオーバーライドして、フレームウィンドウとビューウィンドウの間にスプリットウィンドウが割って入るようにすれば、スプリットウィンドウがフレームウィンドウのクライアント領域を管理できるようになるわけです。これによって、ビューウィンドウの親ウィンドウはスプリットウィンドウになり、スプリットウィンドウの親ウィンドウはフレームウィンドウになります（**図 1-6**）。

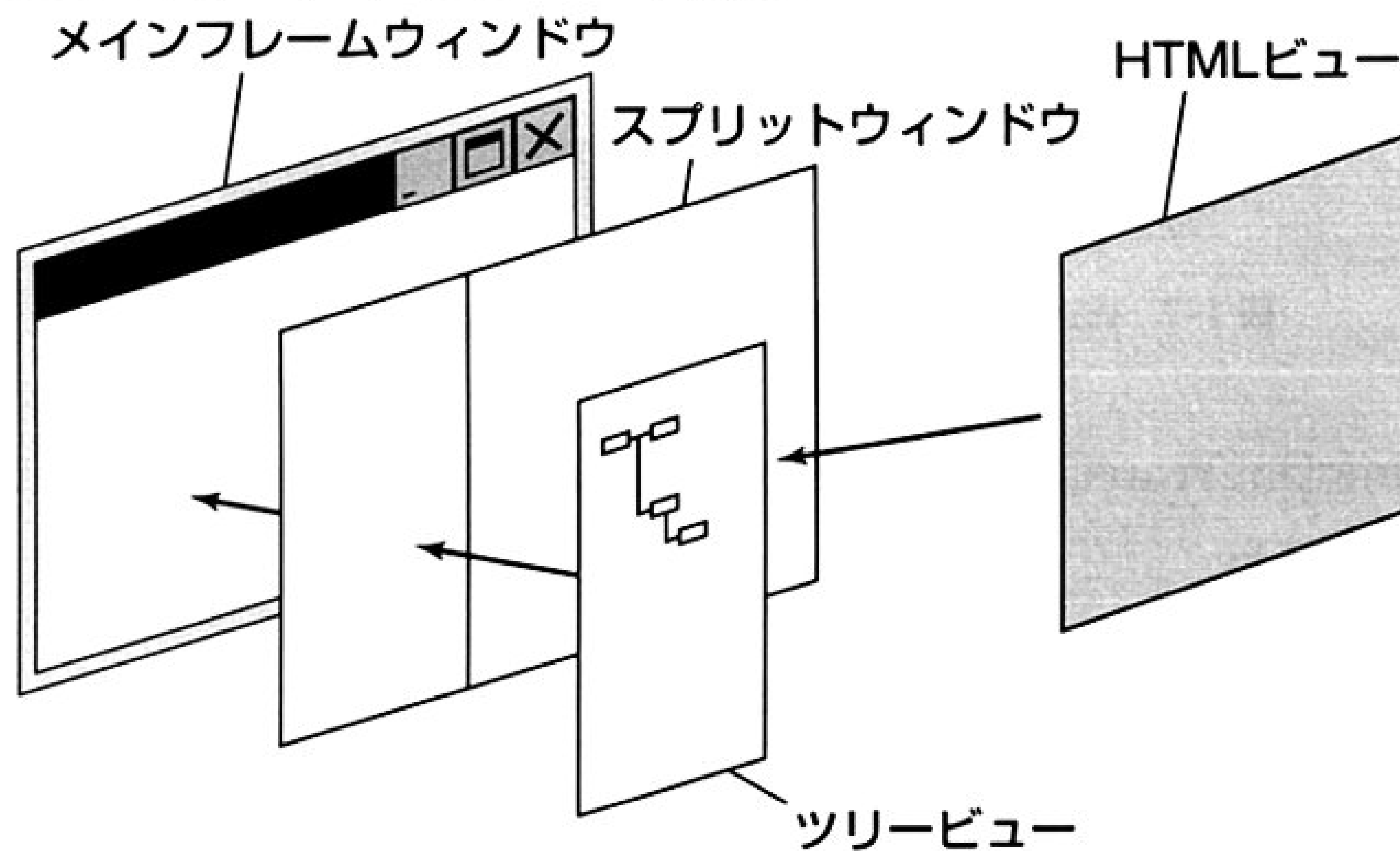


図 1-6　スプリットウィンドウを使ったときのウィンドウの関係

● クライアント領域を乗っ取る！

次に CMainFrame::OnCreateClient メンバ関数を追加して、フレームウィンドウのクライアント領域を乗っ取るためのコードを記述することにします。OnCreateClient メンバ関数は CFrameWnd クラスの仮想メンバ関数ですからオーバーライドすればよいでしょう。オーバーライドには ClassWizard か WizardBar を使います。**表 1-2** に示す指定で CMainFrame::OnCreateClient メンバ関数を作成してください（**図 1-7**）。

クラス	オブジェクト ID	メッセージ
CMainFrame	なし(CMainFrame を選択)	OnCreateClient

表 1-2 OnCreateClient メンバ関数のオーバーライド

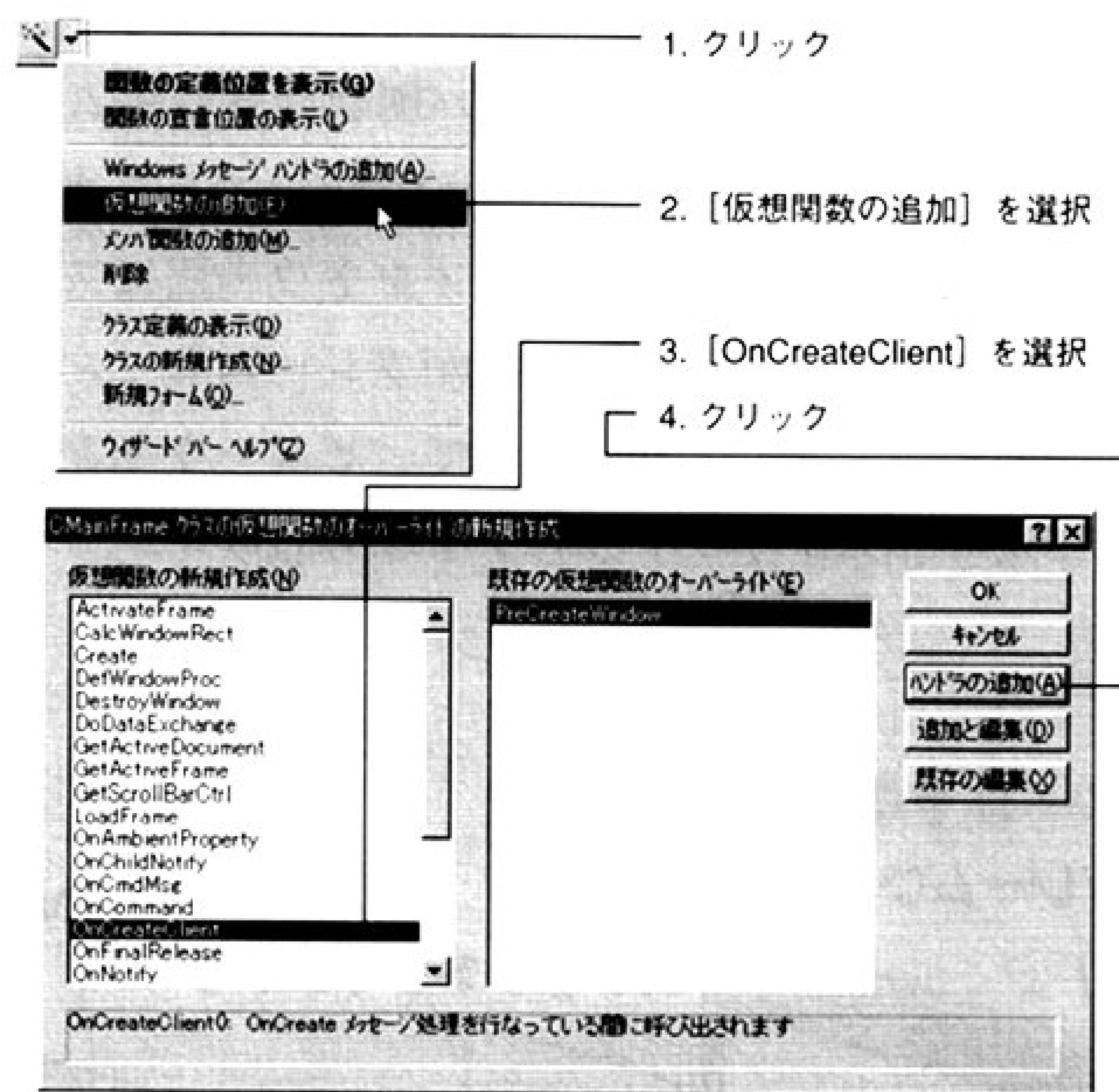


図 1-7 WizardBar を使って OnCreateClient メンバ関数を作る

CMainFrame::OnCreateClient メンバ関数を作ったら、そこに**リスト 1-2** に示すコードを記述してください。

リスト 1-2 CMainFrame::OnCreateClient メンバ関数の実装(MainFrm.cpp)

```
BOOL CMainFrame::OnCreateClient(LPCREATESTRUCT lpcs, CCreateContext* pContext)

{
  if (!m_wndSplitter.CreateStatic(this, 1, 2) ||
    !m_wndSplitter.CreateView(0, 0, RUNTIME_CLASS(CURLTreeView),
      CSize(100, 100), pContext) ||
    !m_wndSplitter.CreateView(0, 1, RUNTIME_CLASS(CBrowserView),
      CSize(0, 0), pContext)) {
    TRACE0("fail to create splitter window");
    return FALSE;
  }
  return TRUE;
}
```

また MainFrm.cpp の先頭に、**リスト 1-3** に示すヘッダファイルの読み込みを追加するのも忘れないでください。

リスト 1-3　ヘッダファイルのインクルード（MainFrm.cpp）

```
#include "URLManDoc.h"
#include "URLTreeView.h"
#include "BrowserView.h"
```

以上でスプリットウィンドウを使うためのコードの追加は終了です。ここまでのコードをコンパイル／実行すれば、**図 1-8** に示すように 2 つのペインが表示されるプログラムが起動します。もっともビューには何も表示されないのでつまらないものですが。

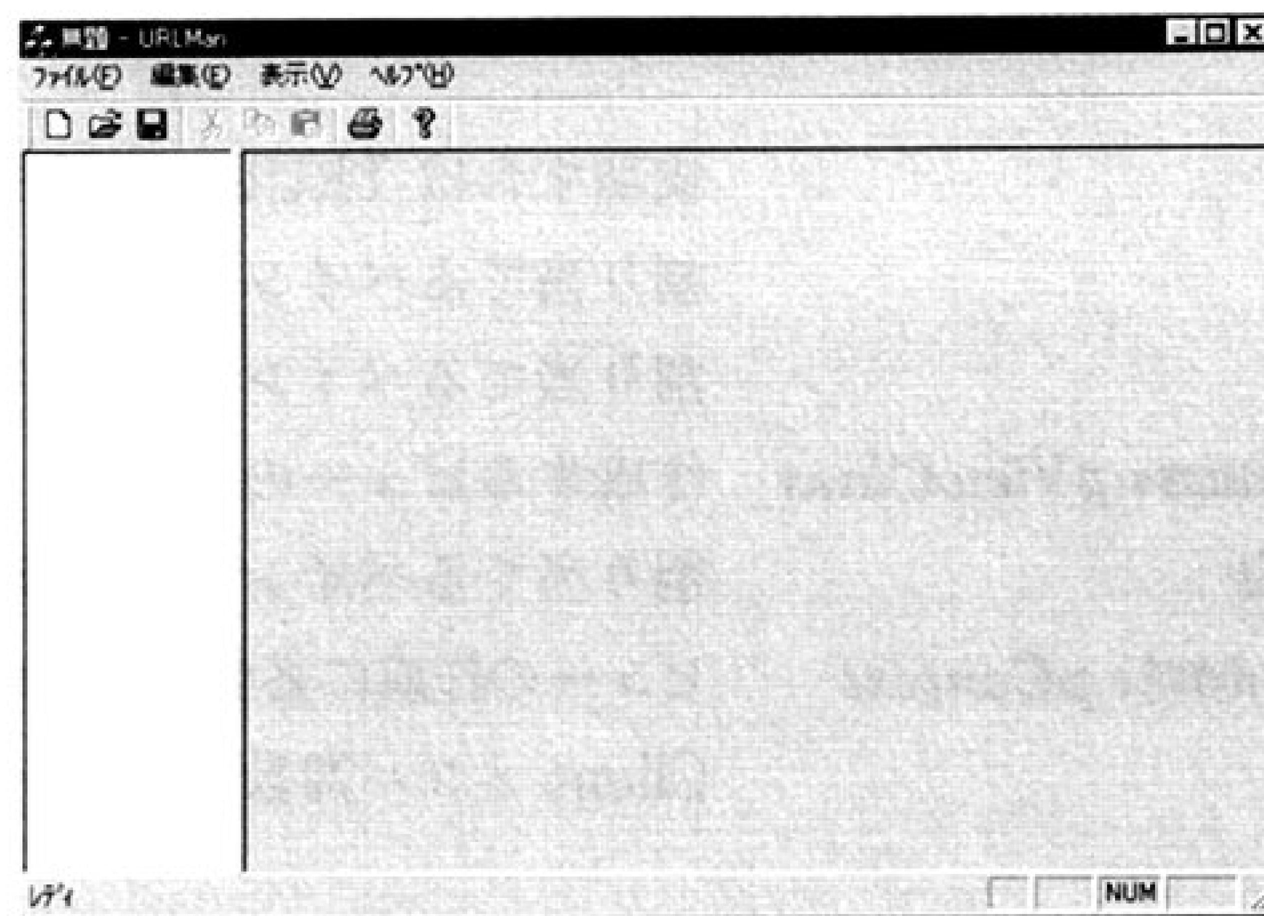


図 1-8　実行画面

コードについて

OnCreateClient メンバ関数に記述したコードでは、次の 2 ステップでスプリットウィンドウを作成しています。

1. ペインの数を定義する（CSplitterWnd::CreateStatic）
2. 各ペインにビュークラスを割り当てる（CSplitterWnd::CreateView）

これら 2 つのメンバ関数は以下のようなことを行ってくれるものです。

BOOL CreateStatic(CWnd* *pParentWnd*, **int** *nRows*, **int** *nCols*,
DWORD *dwStyle* **= WS_CHILD | WS_VISIBLE**,
UINT *nID* **= AFX_IDW_PANE_FIRST)**

返り値	**BOOL**	成功すればTRUE、失敗すればFALSE
引数	**CWnd*** *pParentWnd*	親ウィンドウ（通常はフレームウィンドウ）
	int *nRows*	縦方向に並べるペインの数（16以下を指定）
	int *nCols*	横方向に並べるペインの数（16以下を指定）
	DWORD *dwStyle*	スプリットウィンドウに指定するウィンドウスタイル
	UINT *nID*	子ウィンドウID

BOOL CreateView(int *row*, **int** *col*, **CRuntimeClass*** *pViewClass*,
SIZE *sizeInit*, **CCreateContext*** *pContext***)**

返り値	**BOOL**	成功すればTRUE、失敗すればFALSE
引数	**int** *row*	割り当てるペインの位置（上端のペインが0）
	int *col*	割り当てるペインの位置（左端のペインが0）
	CRuntimeClass* *pViewClass*	作成するビューのクラス
	SIZE *sizeInit*	割り当てるペインの初期サイズ
	CCreateContext* *pContext*	ビューの作成に必要な諸情報（通常はOnCreate Clientメンバ関数の引数をそのまま渡す）

したがって、OnCreateClientメンバ関数では、

```
CreateStatic(this, 1, 2)
```

として、縦に1、横に2つのペインを並べ、次に

```
CreateView(0, 0, RUNTIME_CLASS(CURLTreeView), ……)
```

を実行して、左側のペインにCURLTreeViewクラスを割り当てて、最後に

```
CreateView(0, 1, RUNTIME_CLASS(CBrowserView), ……)
```

として右側のペインにCBrowserViewクラスを割り当てていたというわけです。

●コンポーネントギャラリ

ところで、本節で述べたスプリットウィンドウは、ほとんど手直しせずに、そのままの方法で他のアプリケーションにも組み込むことができます。こうした再利用可能なモジュールを保存しておいて、次回からは手間暇をかけずに利用したいと思うのが人情というものです。実はVisual C++にはこうしたモジュールを管理して、プロジェクトに簡単に組み込む機構が用意されています。この機構は**コンポーネントギャラリ**と呼ばれ、メニューから［プロジェクト］－［プロジェクトへ追加］－［コンポーネントおよびコントロール］を実行することにより起動することができます。

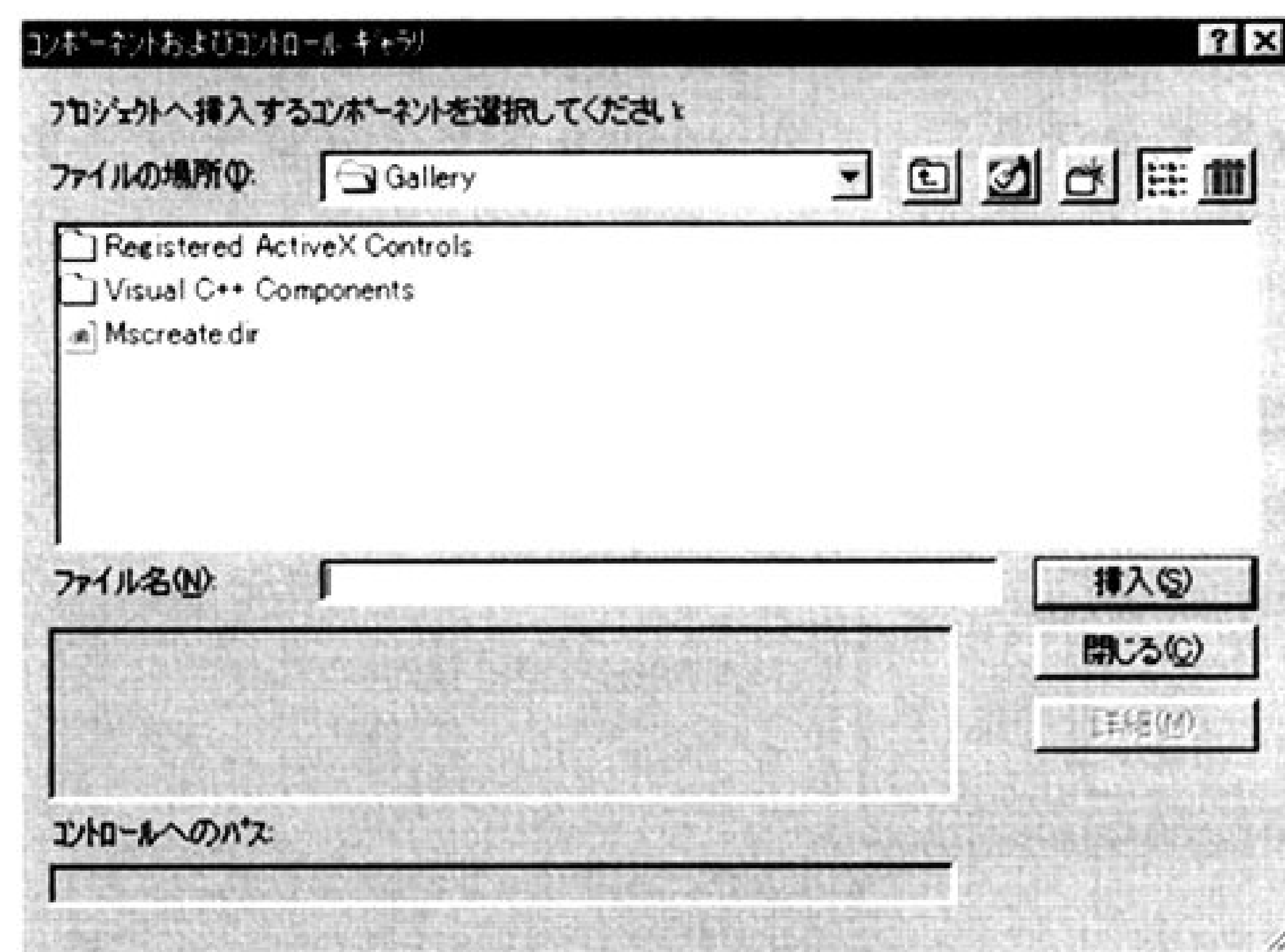


図1-9　コンポーネントギャラリ

「あらかじめ用意されているコードを自分のプロジェクトに利用する」という点ではAppWizardが生成するスケルトンに通じるものがありますが、コンポーネントギャラリは利用するタイミングがAppWizardとは異なります。AppWizardはプロジェクトをスタートさせる最初の段階でしか利用することができませんが、コンポーネントギャラリはプロジェクトの開発段階の**いついかなるときにでも利用することができます**。

汎用的なモジュールを用意するならば、オブジェクトファイルを別に用意しておいて、あとからそれをリンクするだけでもよいのではないかと考える人もいるかもしれませんが、それでは不可能なことも可能にするのがコンポーネントギャラリです。たとえば、本節で作業を行ったスプリットウィンドウの作成は、#include文を挿入したり、メンバ変数を追加したりと、別ファイルを用意しておいてあとからそれをリンクするのでは実現できない作業です。ところがコンポーネントギャラリは、現在のソースコードの状態を把握して、コードを適切な形で挿入してくれるのです。

なお、コンポーネントギャラリにも［スプリットバー］なるコンポーネントが用意されていますが、これを本節で説明したコードを記述する代わりに使用することはできません(残念)。実はスプリットウィンドウには「静的スプリットバーを使用するもの」と「動的ス

プリットバーを使用するもの」の2種類があります。そして、本節で解説したのは「静的スプリットバーを使用するもの」、コンポーネントとして用意されているのは「動的スプリットバーを使用するもの」というわけで、コンポーネントギャラリを使用してもあとでコードの変更が必要になることから、ここでは直接手作業でスプリットウィンドウを作成したのです。

エクスプローラスタイル

本章での作業はすでに終えていると思いますから、意図的に解説を避けてきた機能を紹介しましょう。実はAppWizardの新機能を使えば、本章で解説した作業の大半をAppWizardに自動的に行わせることができるのです。SDI/MDI形式のアプリケーションを作成する場合、AppWizardのステップ5で、［Windowsエクスプローラスタイル］という選択肢が表示されます。

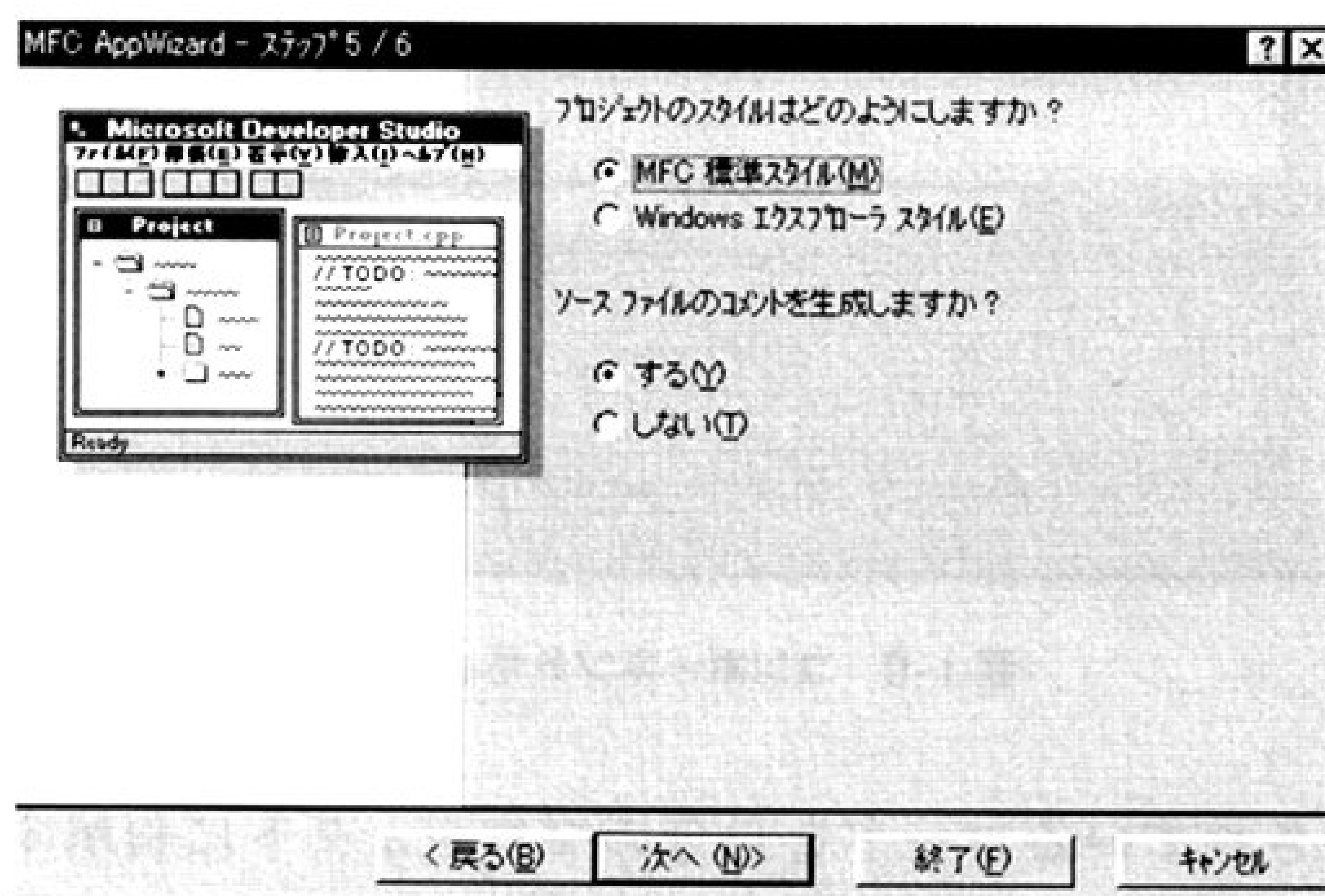


図1-10　Windowsエクスプローラスタイル

これを選択すると、まさに本章で皆さんが行った作業がほぼそのままAppWizardによって行われてしまいます。つまり、メインフレームウィンドウにスプリットウィンドウを埋め込み、左のペインにはツリービューを、右のペインにはHTMLビューをそれぞれ用意する作業がすべてAppWizardによって自動化されます。また、右のペインに割り当てるビューについては、CViewクラスの派生クラスからユーザーが選択することができます。

このように便利な機能ではありますが、その内容を知らなければ活用することは難しいので、本章では利用せずに済ませました。しかし生成されたコードの内容はすでに本章で解説しましたから、次からは大いに利用してください。

以上で本章の解説を終わりにして、次章では、実作業に入る前の準備をさらに進め、またこれに必要となる知識を少々手に入れることにしましょう。

2 プログラム作成の前準備

1章では、スケルトンを作成し、ビュークラスを1つ追加、さらにスプリットウィンドウを作成しましたが、1つのアプリケーションとして完成するまでには、まだまだ足りないものが山ほどあります。ビュークラスの操作をメインディッシュとして3章以降で味わうことにして、ひとまず本章ではURLリストを管理するドキュメントクラスを先に実装してしまうことにします。

ドキュメントクラスの実装方法はすでに第3部のMMViewで解説していますが、URLマネージャとMMViewでは少しだけ違う点があります。それは、MMViewが扱うデータファイルがバイナリ形式であったのに対して、URLマネージャではテキスト形式であるということです。テキスト形式のデータファイルであっても、データファイルを読み書きするためにCObject::Serialize仮想メンバ関数をオーバーライドし、CArchiveオブジェクトを使用する点では変わりありませんが、CArchiveオブジェクトの使用方法が異なります。

2.1 データファイルのフォーマット

第4部の主役はあとに控えしビュークラスですが、ビュークラスと対をなすドキュメントクラスなくしてはドキュメント－ビュー・アーキテクチャを使ったアプリケーションは語れません。しかしドキュメントクラスはここではあくまでも脇役ですから、次のように非常に単純なものとします。

- **登録できるURLの個数は1000件に限る**
- **URLの編集はできない**
- **URLの保存はできない**
- **ディレクトリ階層は1段階に限る**

上記の制限を設けることによって、ドキュメントクラスはかなり単純なものになります。

ドキュメントクラスの実装を始める前に、まずlsurlツールが作成するテキスト形式のデータファイルのフォーマットについて説明しておきましょう。データファイルに含まれる情報には、

- **フォルダ名**
- **URLとWebページのタイトル**

の2種類があります。さらに、各URLがどのフォルダに含まれているのかを示す必要もあります。

リスト2-1にデータファイル例を示します。これは、**図2-1**のように登録されたお気に入りをlsurlツールでテキストファイルに変換したものです。

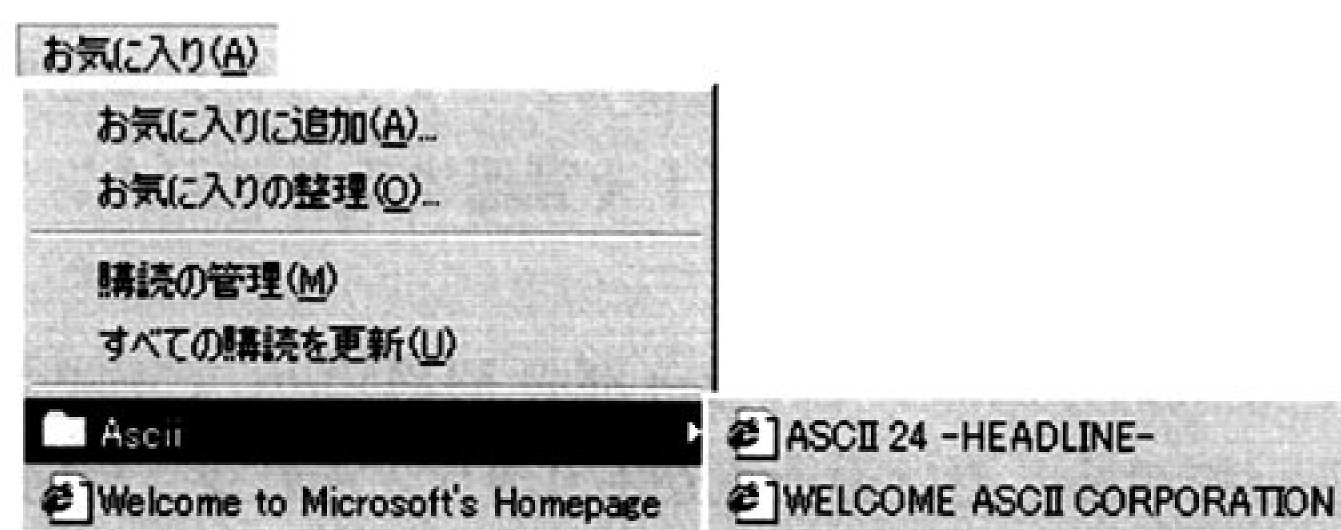


図2-1 お気に入り例

リスト2-1 データファイル例

```
+ASCII
http://www.ascii.co.jp/ascii24/ ASCII 24 -HEADLINE-
http://www.ascii.co.jp/ WELCOME ASCII CORPORATION
-
http://www.microsoft.com/ Welcome to Microsoft's Homepage
```

リスト2-1を先頭から見ていきましょう。1行目のように、行頭が「+」で始まっている行は、ここからフォルダが始まっていることを意味し、続く文字列がフォルダ名を表しています。フォルダ名の終端は行末で表しています。

次に2行目ですが、これがURLと対応するページタイトル文字列を表しています。URLは行頭から始まり、ページタイトルは行末で終わります。また、その区切れ目には1文字のタブ文字('¥t')を挿入しています。つまり、URLとページタイトルを含む行は、次の構造をしています。

<行頭><URL><タブ文字><ページタイトル><行末>

3 行目は 2 行目と同じく URL を含む行ですから飛ばして、4 行目を見ると、「－」だけが含まれています。これは、フォルダの終了を表しています。この例では 1 行目の「＋ASCII」から 4 行目の「－」までが、フォルダ ASCII に含まれていることを表しているわけです。なお、URL マネージャではフォルダの階層は 1 段階までに限っているので、フォルダの中にフォルダを作るようなデータは認めないことにします。

次の 5 行目は同じく URL を含む行ですが、4 行目でフォルダ ASCII は終了しているので、この Microsoft のホームページへの URL は、フォルダに含まれず、トップレベルに表示されることになります。

以上をまとめると次のようになります。

1. 行頭が「＋」ならば、フォルダの開始を意味する
2. 行頭が「－」ならば、フォルダの終了を意味する
3. どちらでもなければ、URL を含む行として扱う

2.2 URL情報の格納場所を用意する

それでは、ドキュメントクラスの実装を始めることにしましょう。ドキュメントクラスが担当する役割は、第 3 部で解説した内容と極端に違うところはありません。復習のつもりで読み進めてください。

まず、データファイルから読み込んだ URL リストを格納するために、メンバ変数をドキュメントクラスに追加します。データファイルから読み込んだ URL はこれから作成する CURLEntry クラスに格納し、ドキュメントクラスにはこの CURLEntry クラスの配列をメンバ変数として追加することにします。

データファイルには 1 行につき URL とページタイトルが組になって格納されていますから、CURLEntry クラスは**リスト 2-2** のように設計します。CURLEntry クラスの定義は、このクラスを使うのはドキュメントクラスだけですから、CURLManDoc クラスが定義されている URLManDoc.h ファイルの先頭部分で行えばよいでしょう。なお、URL マネージャでは第 3 部で解説したように、CObject クラスによるシリアライズのしくみを利用したファイルの読み書きは行わないので、CURLEntry クラスは基底クラスを持たない、通常のクラスとして定義しています。

リスト2-2　CURLEntryクラス(URLManDoc.h)

```
class CURLEntry
{
private:
  CString m_strTitle;
  CString m_strURL;

public:
  CURLEntry() { m_strTitle = ""; m_strURL = ""; }
  CURLEntry(CString& t, CString& u) {
    m_strTitle = t; m_strURL = u; }
  CString& getTitle() { return m_strTitle; }
  CString& getURL() { return m_strURL; }
};
```

CURLEntryクラスのメンバ変数には、m_strTitleメンバ変数にページタイトルが、m_strURLにURLがそれぞれ文字列として格納されますが、フォルダの情報を格納する場合には、特殊な文字列をセットします。行頭が「＋」で始まるフォルダの開始を意味する行を読み込んだときには、普段はURLが格納されるm_strURLメンバ変数に文字列「dir」を格納し、m_strTitleメンバ変数にフォルダ名を格納します。また、「－」だけからなるフォルダの終了を意味する行を読み込んだときには、m_strURLメンバ変数に文字列「up」を格納し、m_strTitleメンバ変数には空文字列を格納します。つまり、URLかフォルダ情報かにかかわらず、データファイルの1行がそのまま1つのCURLEntryオブジェクトに対応することになります。

以上のように定義したCURLEntryクラスの配列を**リスト2-3**のように、CURLManDocクラスのm_arrayURLメンバ変数として追加します。また、いくつの配列要素を使ったのかを格納するために、int型のm_nIndexメンバ変数も同時に追加します。

リスト2-3　m_arrayURLメンバ変数の追加(URLManDoc.h)

```
#define MAX_URL 1000
// 格納できる URL の個数

class CURLManDoc : public CDocument
{
...
// アトリビュート
public:
  CURLEntry* m_arrayURL[MAX_URL];
  // データファイルから読み込んだ URL とフォルダ情報を格納する

  int m_nIndex;
```

```
    // m_arrayURLメンバ変数で使われている要素数
    // m_arrayURLメンバ変数はフォルダ情報の格納にも使われるので、
    // 格納されているURLの個数ではない
  ...
};
```

m_arrrayURLメンバ変数はCURLEntryクラスへのポインタの配列です。この配列のサイズはMAX_URLとして1000に定義されているので、これがURLマネージャで扱えるデータサイズの上限となります。ところで、本章の冒頭でURLマネージャが扱えるURLの個数は1000個に限ると述べましたが、上述したようにフォルダ情報を格納するためにもCURLEntryオブジェクトを使用するので、フォルダ情報が含まれる場合には、格納できるURLの個数は1000－フォルダの個数×2（開始と終了で2つのCURLEntryオブジェクトを使用するため）になります。

メンバ変数を用意したところで、データファイルの読み込みを行う前に、m_arrayURLメンバ変数の初期化と後始末の処理を先に実装してしまいましょう。

まず初期化ですが、これはCURLManDocクラスのコンストラクタで行います。AppWizardによってすでに空のコンストラクタが用意されていますから、**リスト2-4**のように3行追加してください。内容はm_arrayURLメンバ変数の配列要素をすべてNULL値にし、データ数が0であることを示すためにm_nIndexメンバ変数に0を代入するだけの、簡単なものです。

リスト2-4 CURLManDocクラスのコンストラクタ（URLManDoc.cpp）

```
CURLManDoc::CURLManDoc()
{
  for (int i = 0; i < MAX_URL; i++)
    m_arrayURL[i] = NULL;
  m_nIndex = 0;
}
```

次に後始末ですが、今度はCURLManDoc::DeleteContents仮想メンバ関数で行います。

初期化をコンストラクタで行いながら、後始末はデストラクタで行わないのは不思議に思われるかもしれませんが、これには理由があります。第3部のMMViewのようなMDIアプリケーションならばドキュメントクラスのデストラクタとDeleteContents仮想メンバ関数のどちらに記述してもかまわないのですが、URLマネージャのようなSDIアプリケーションでは、DeleteContents仮想メンバ関数を実装しなければなりません。ここでは、SDIアプリケーションに限ってその理由を説明しましょう。

DeleteContentsメンバ関数はCDocumentクラスで定義されている仮想メンバ関数です。このメンバ関数はアプリケーションの終了時や、［ファイル］－［新規作成］を実行して編集中のドキュメントをクリアするときなど、ドキュメントの内容を破棄しなければならないときに呼び出されます。アプリケーションの終了時にはドキュメントクラスのデストラクタも呼び出されるのですが、問題はドキュメントを新規作成したときです。このときには、CDocument::DeleteContents仮想メンバ関数だけが呼び出され、ドキュメントクラスのデストラクタは呼び出されません。つまり、SDIアプリケーションではアプリケーションの実行時にドキュメントオブジェクトが作成され、アプリケーションが終了するまでずっとこのドキュメントオブジェクトが使い続けられるのです。したがって、CDocument::DeleteContents仮想メンバ関数を実装しなければ、ドキュメントを新規作成しても、古いデータが残ったまま次の編集作業が始まってしまうというわけです。

リスト2-5にCURLManDoc::DeleteContents仮想メンバ関数の内容を示します。CURLManDoc::DeleteContents仮想メンバ関数はClassWizardまたはWizardBarを使ってオーバーライドし、CURLManDocクラスに追加してください。

クラス	オブジェクトID	メッセージ
CURLManDoc	なし（CURLManDocを選択）	DeleteContents

表2-1 CURLManDoc::DeleteContents仮想メンバ関数をオーバーライドする

リスト2-5 CURLManDoc::DeleteContents仮想メンバ関数（URLManDoc.cpp）

```
void CURLManDoc::DeleteContents()
{
  // m_arrayURLの内容を削除すると同時に、CURLManDoc::CURLManDoc()と
  // 同じように、ドキュメントの初期化も行う
  for (int i = 0; i < m_nIndex; i++) {
    delete m_arrayURL[i];
    m_arrayURL[i] = NULL;
  }

  m_nIndex = 0;

  CDocument::DeleteContents();
}
```

2.3 テキストファイルを読み込む

URL 情報を格納する準備が整ったので、最後にテキスト形式のデータファイルを読み込む方法について解説しましょう。

MFC が採用しているドキュメント－ビュー・アーキテクチャでのファイルの入出力はシリアライズと呼ばれ、Serialize 仮想メンバ関数と CArchive クラスがキーとなることはすでに第 3 部で取り上げたとおりです。ただし、シリアライズによって入出力を行うデータファイルはバイナリ形式のものであるため、テキスト形式のデータファイルを扱う URL マネージャでは少し違った方法が必要になります。

［ファイル］－［開く］など、ファイルの入出力を行うメニューコマンドを実行すると、CObject::Serialize 仮想メンバ関数が呼び出されるので、この仮想メンバ関数にコードを追加する点では変わりありませんし、CArchive クラスを使う点でも同じです。ただし、CArchive クラスの<<演算子や>>演算子は使いません。代わりに使用するのが、CArchive::ReadString メンバ関数です。このメンバ関数はデータファイルをテキスト形式として扱い、キャリッジリターン（'¥r'）とラインフィード（'¥n'）で表される行末までを引数に指定された CString 型オブジェクトに読み込みます。

Bool CArchive::ReadString(CString& *rString*)

返り値	**BOOL**	正常終了すると TRUE、異常終了（EOF を含む）すると FALSE を返す
引数	**CString&** *rString*	rString に 1 行読み込まれる。行末の改行は取り除かれる

CArchive::ReadString 関数を使えば、返り値でファイルの終端を調べることができますから、細かなエラーハンドリングを省けば、次のコードでテキストファイルを 1 行ずつ、最後まで読み込む処理が記述できます。

```
CArchive ar;  // 初期化済みとする
CString line; // データを読み込むバッファとして使う

while (ar.ReadString(line)) {
  // line に 1 行分のデータが読み込まれている
}
```

したがって、あとは変数 line に読み込まれたデータを解析して、フォルダ情報か URL 情報かを判断して、処理すればよいことになります。

ファイルの入出力処理は CURLManDoc::Serialize 仮想メンバ関数に記述しますが、わかりやすくするために、データの読み込み用に独立したメンバ関数を CURLManDoc クラス

に追加しましょう。名前はCURLManDoc::ReadFavoritesメンバ関数として、次のようにして作成してください。

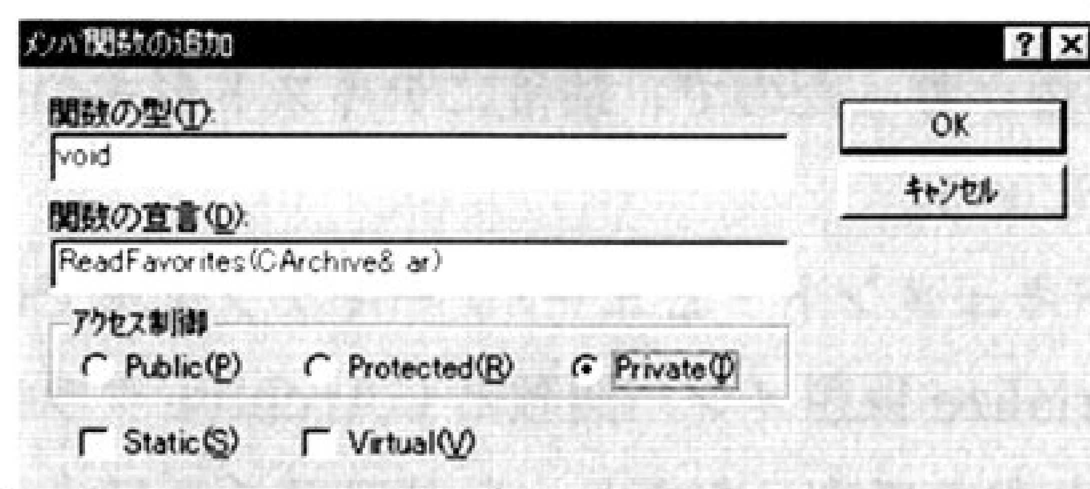


図 2-2　CURLManDoc::ReadFavoritesメンバ関数の作成

CURLManDoc::ReadFavoritesメンバ関数のスケルトンが作成されたら、**リスト 2-6**のようにコードを入力してください。

リスト 2-6　お気に入りを読み込むためのコード(URLManDoc.cpp)

```
void CURLManDoc::Serialize(CArchive& ar)
{
  if (ar.IsStoring())
  {
    // TODO: この位置に保存用のコードを追加してください
  }
  else
  {
    ReadFavorites(ar);
  }
}

void CURLManDoc::ReadFavorites(CArchive& ar)
{
  CString line;   //データ読み込み用バッファ
  CString url;    //line から抜き出した URL を格納する
  CString title;  //line から抜き出したページタイトルを格納する

  int pos;

  while (ar.ReadString(line) && m_nIndex < MAX_URL) {
    if (line[0] == '+') {
      //ディレクトリの開始
      title = line.Mid(1, line.GetLength() - 1);
      m_arrayURL[m_nIndex++] =
        new CURLEntry(title, CString("dir"));
    } else if (line[0] == '-') {
      //ディレクトリの終了
      m_arrayURL[m_nIndex++] =
        new CURLEntry(CString(""), CString("up"));
```

```
    } else if ((pos = line.Find('¥t')) != -1) {
      // URL
      url = line.Left(pos);
      title = line.Mid(pos + 1, line.GetLength() - pos - 1);
      m_arrayURL[m_nIndex++] =
        new CURLEntry(title, url);
    }
  }
}
```

CURLManDoc::ReadFavorites メンバ関数での処理はほとんどが単純な文字列操作にすぎません。CArchive::ReadString メンバ関数で読み込んだ1行分の文字列データから必要な部分を切り出し、新規に作成したCURLEntryオブジェクトへと格納しているだけです。

こうしてm_arrayURLメンバ変数に格納されたデータがどのようにツリービューコントロールへと反映されるのか、それはCURLTreeViewクラスの実装になりますから、これでドキュメントクラスの解説は終わりです。最後に、ここで使われているCStringクラスのメンバ関数をまとめておきましょう。

TCHAR CString::operator[](int *nIndex*)

返り値	TCHAR	*nIndex* に指定した位置の文字を返す
引数	int *nIndex*	文字列の中から取り出す文字の位置を0から始まるインデックスで指定する

int CString::GetLength()

返り値	int	文字列の長さを返す。単位はバイト

CString CString::Mid(int *nFirst*, int *nCount*)

返り値	CString	引数で指定した範囲の文字列を返す
引数	int *nFirst*	抜き出す範囲の先頭位置を0から始まるインデックスで指定する
	int *nCount*	*nFirst* で指定した位置から抜き出す文字列の長さをバイト単位で指定する

CString CString::Left(int *nCount*)

返り値	CString	引数で指定した範囲の文字列を返す
引数	int *nCount*	文字列の先頭からnCount文字数だけ取り出す

3 ツリービューコントロールを使ってみよう

さて、いよいよURLマネージャの肝であるツリービューコントロールの解説を始めます。ツリービューコントロールはエクスプローラ、コントロールパネルのシステムアプレットをはじめとして、Windows 95以降ではいたるところで使われているコントロールです(**図3-1**)。データをただズラズラと一列に並べるのに比べて、階層的にデータをまとめることができるツリービューコントロールはわかりやすいユーザーインターフェイスをユーザーに提供できる、応用範囲の広いコントロールです。

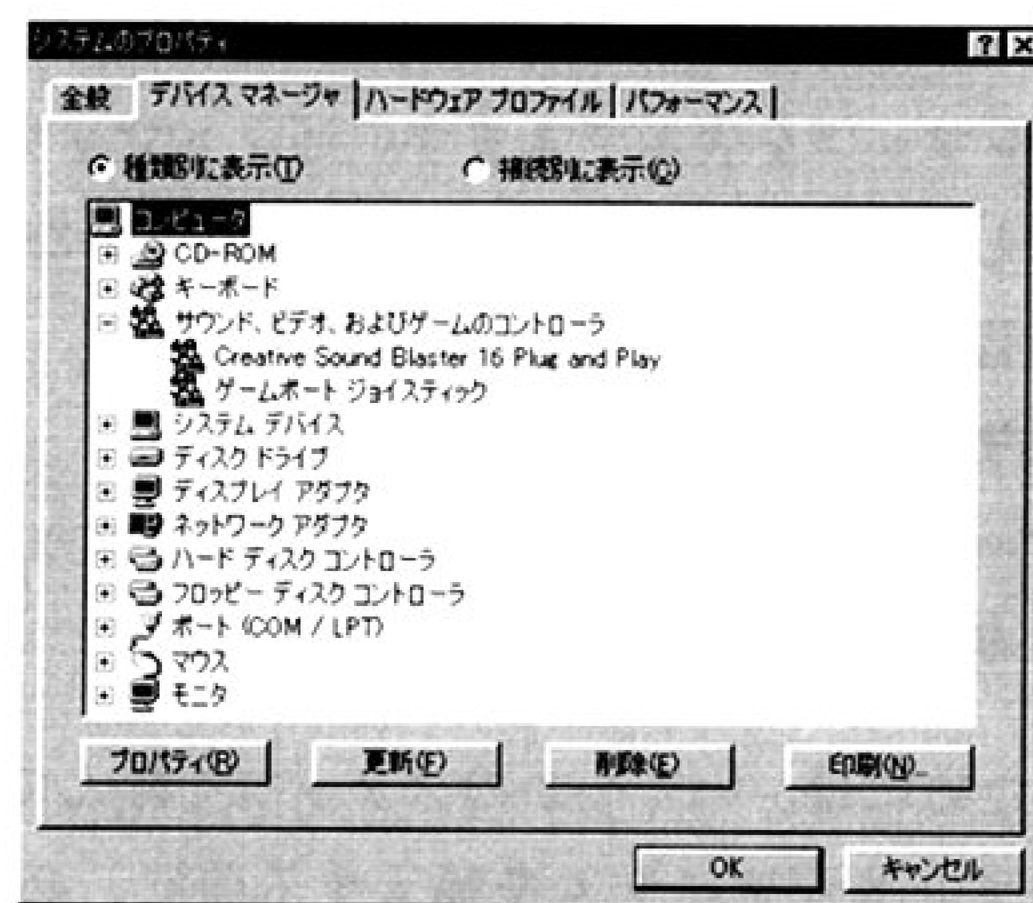


図3-1 ツリービューコントロール

本章ではURLマネージャの左側のペインに割り当てられる、CURLTreeViewクラスを題材にして、ツリービューコントロールの使い方について解説します。

ところで第4部の冒頭でも述べたように、ツリービューコントロールの使い方は複雑であるため、DDXを使った操作は行いません。したがって、本章ではDDXという言葉はいっさい登場しません。MFCを使ったWindowsプログラミングというと、DDXを使えばよいと考えるかもしれませんが、それだけではできないことも数多く存在するのです。DDXを使わずに、特定のコントロールを細かく制御する方法を感じ取ってください。

3.1 ツリービューコントロールの機能

ツリービューコントロールをおおざっぱに説明すれば、「拡張リストボックスコントロール」といったところでしょう。アイテムをリストアップして、そこからアイテムを選択して……、というユーザーインターフェイスを作る目的に変わりはありません。どのあたりが「拡張」されているかというと、次の３点があげられます。

- **アイテムを階層化して表示できる**
  ツリーコントロールの名前が示すように、アイテムをツリー状に表示することができます。リストボックスをファイルしか作れないファイルシステムに例えれば、ツリーコントロールではフォルダを作れるようになったということです。このフォルダに相当するアイテムをダブルクリックすることによって、それよりも深い階層に位置するアイテムを表示したり、隠したりすることができます。
- **アイテムごとにビットマップを割り当て、文字列の前に表示できる**
  リストボックスでは文字列しか表示することができず、ビットマップなどを表示したければオーナードローと呼ばれるテクニックを使わなければなりませんでした。これは面倒な作業でしたが、ツリービューは、文字列と同時にビットマップを表示するできるので、簡単にビットマップを表示できます。ビットマップはアイテムごとに異なったものを指定できますし、選択されている／選択されていないなど、状態に応じて表示するビットマップを複数使用することもできます。
- **ドラッグ&ドロップのサポート**
  ツリービューコントロールにはアイテムをドラッグ&ドロップによって操作する機能があります。ただし、本書では扱いません。

こうした機能を利用するためには、いくつかのステップを踏んでツリービューコントロールを操作する必要があります。まずアイテムを挿入する前に、必要に応じて次の２つの初期化処理を行っておきます。

- ツリービューコントロールのスタイルの設定
- 各アイテムの左側に表示するビットマップの準備（イメージリストの作成）
  ツリービューにビットマップを表示しない場合にはこの作業は不要です。

初期化処理を行うと、アイテムをツリービューコントロールに登録できるようになります。アイテムを登録したら、今度はアプリケーションのユーザーがツリービューコントロールを操作できるようになるので、マウスのクリックなどの処理を考慮するようにしなければなりません。

本章では、これらの処理を順を追って解説していきます。まずは上述の2つの初期化処理の方法について。ついでアイテムの登録の仕方。最後にユーザーがアイテムをクリックしたときの処理方法について解説します。

3.2 初期化1：ツリービューコントロールのスタイルを設定する

ツリービューコントロールには、**表3-1** および**図3-2** に示すようなスタイルを設定することによって、見せ方や機能をカスタマイズできます。

スタイル	効果
TVS_HASLINES	親アイテムと子アイテムを結ぶ線が引かれる
TVS_LINESATROOT	最上位階層に位置するアイテム同士を結ぶ線が引かれる
TVS_HASBUTTONS	子アイテムを持つアイテムの前にボタンが表示される
TVS_EDITLABELS	各アイテムのラベルを編集可能にする

表3-1 ツリービューコントロールのスタイル

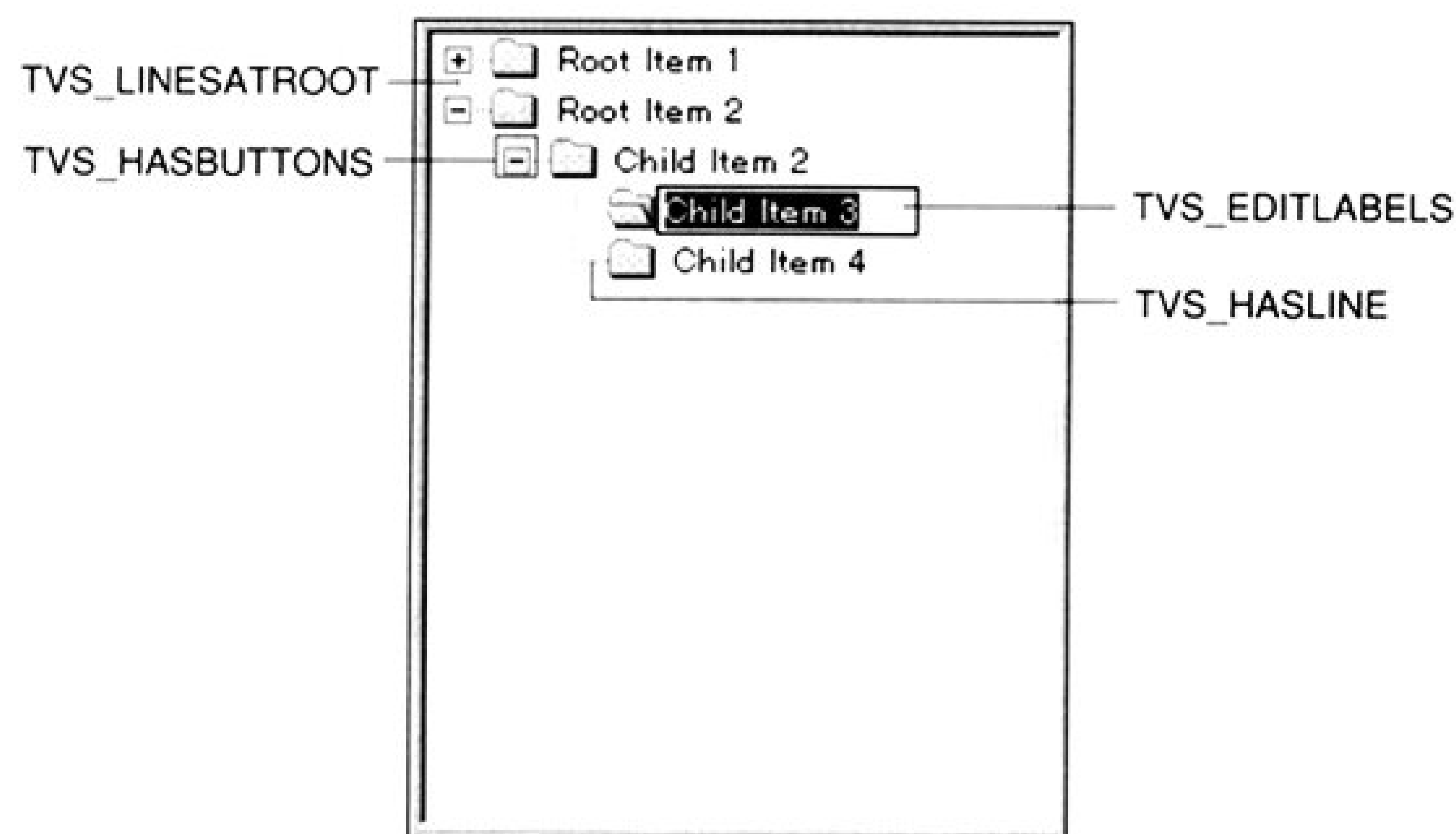


図3-2 ツリービューコントロールのスタイル

デフォルトの状態では何のスタイルも指定されていないので、アイテム(文字列)だけが表示されるという、ちょっとお寒い状態です。少なくともTVS_HASLINESとTVS_HASBUTTONSは設定しておくとよいでしょう。ただし、これらのスタイルには次の依存関係があることに注意してください。

- TVS_HASLINESがないとTVS_LINESATROOTは機能しない
- TVS_LINESATROOTがないと最上位階層のアイテムにはボタンが表示されない

つまり、TVS_LINESATROOTを設定するときにはTVS_HASLINESを、TVS_HASBUTTONSを設定するときにはTVS_LINESATROOTをそれぞれかならず設定する必要があります。

これらツリービューコントロールのスタイルを設定するには、CWnd::PreCreateWindow仮想メンバ関数をオーバーライドします。これは通常のウィンドウスタイルを設定する方法と同じです。実際には、ウィンドウであればどのようなものでもPreCreateWindow仮想メンバ関数をオーバーライドすることによって、ウィンドウスタイル(WS_~)を設定できるようになっています。

CWnd::PreCreateWindow仮想メンバ関数はウィンドウが作成される寸前にフレームワークによって呼び出される関数です。その引数にはウィンドウを作成するために用いられるパラメータが含まれているので、PreCreateWindow仮想メンバ関数をオーバーライドして引数の値を変更することによって、デフォルトの状態(のウィンドウスタイルやウィンドウサイズなど)とは異なるウィンドウを作ることが可能になります。CWnd::PreCreateWindowメンバ関数の書式は次のようなものです。

BOOL CWnd::PreCreateWindow(CREATESTRUCT& *cs***)**

返り値	**BOOL**	ウィンドウの作成を継続する場合には0以外、中止する場合には0
引数	**CREATESTRUCT&** *cs*	ウィンドウを作成するのに必要なパラメータ

PreCreateWindow関数にはCREATESTRUCT&型の引数csが渡されるので、今述べたようにPreCreateWindow関数内でcsを操作することで望みのウィンドウスタイルを設定することができます。CREATESTRUCT構造体のメンバ変数の一部を**表3-2**に示します。これ以外のメンバについては、フレームワークの内部を詳しく理解するまでは手を付けない方が賢明でしょう。

型	メンバ変数名	用途
int	cy	ウィンドウサイズ(縦)
int	cx	ウィンドウサイズ(横)
int	y	ウィンドウの表示位置(Y軸)
int	x	ウィンドウの表示位置(X軸)
LONG	style	ウィンドウスタイル
LPCSTR	lpszName	ウィンドウの名前
LPCSTR	lpszClass	ウィンドウクラスの名前
DWORD	dwExStyle	ウィンドウ拡張スタイル

表3-2 CREATESTRUCT構造体のメンバ

ここではウィンドウスタイルを変更したいので、style メンバ変数を書き換えることにします。CURLTreeView::PreCreateWindow 仮想メンバ関数はすでに AppWizard によって作成されていますから、**リスト 3-1** でアミがかかっている行を追加するだけでスタイルを設定できます。URL マネージャでは TVS_HASLINES、TVS_LINESATROOT、TVS_HASBUTTONS の 3 つのスタイルを設定しています。

リスト 3-1　ウィンドウスタイルの設定（FriendTreeView.cpp）

```
BOOL CURLTreeView::PreCreateWindow(CREATESTRUCT& cs)
{
    cs.style |= (TVS_HASLINES | TVS_LINESATROOT | TVS_HASBUTTONS);
    //3 つのスタイルを追加する

    return CTreeView::PreCreateWindow(cs);
}
```

ウィンドウスタイルを設定するには、**リスト 3-1** で行っているように必要なスタイルを論理和で結び、さらに cs.style に論理和で加えるようにしてください。

3.3 初期化2:ビットマップの準備

ツリービューコントロールの見た目はスタイルだけではなく、ビットマップによってもガラッと変わります。アイテムの左側にビットマップを表示するかどうかはスタイルではなく、あとで述べるように、アイテムを挿入するときのパラメータによって決まります。つまり、ビットマップの表示、非表示はツリービューコントロール全体に対する設定ではなく、各アイテムに対する設定です。またアイテムごとに異なるビットマップを指定することもできます。

ビットマップをツリービューコントロールに登録するには、以下のような作業が必要になります。

1. ビットマップリソースの用意
2. イメージリストオブジェクトの用意
3. ビットマップをイメージリストに登録する
4. イメージリストをツリービューコントロールに登録する

MFC にはビットマップを扱うクラスとして CBitmap クラスがありますが、ツリービューコントロールでは複数のビットマップを配列のような形でまとめて扱うことができる**イメージリスト**というものを使用します。ツリービューコントロールでは、多くの場合アイテム

ごとやアイテムの状態（選択されているか否か、フォルダが展開されているか否かなど）ごとにビットマップを使い分けることになるので、複数のビットマップをまとめて管理できると便利だからです。イメージリストを扱うためのクラスとしては、CImageList クラスが用意されています。

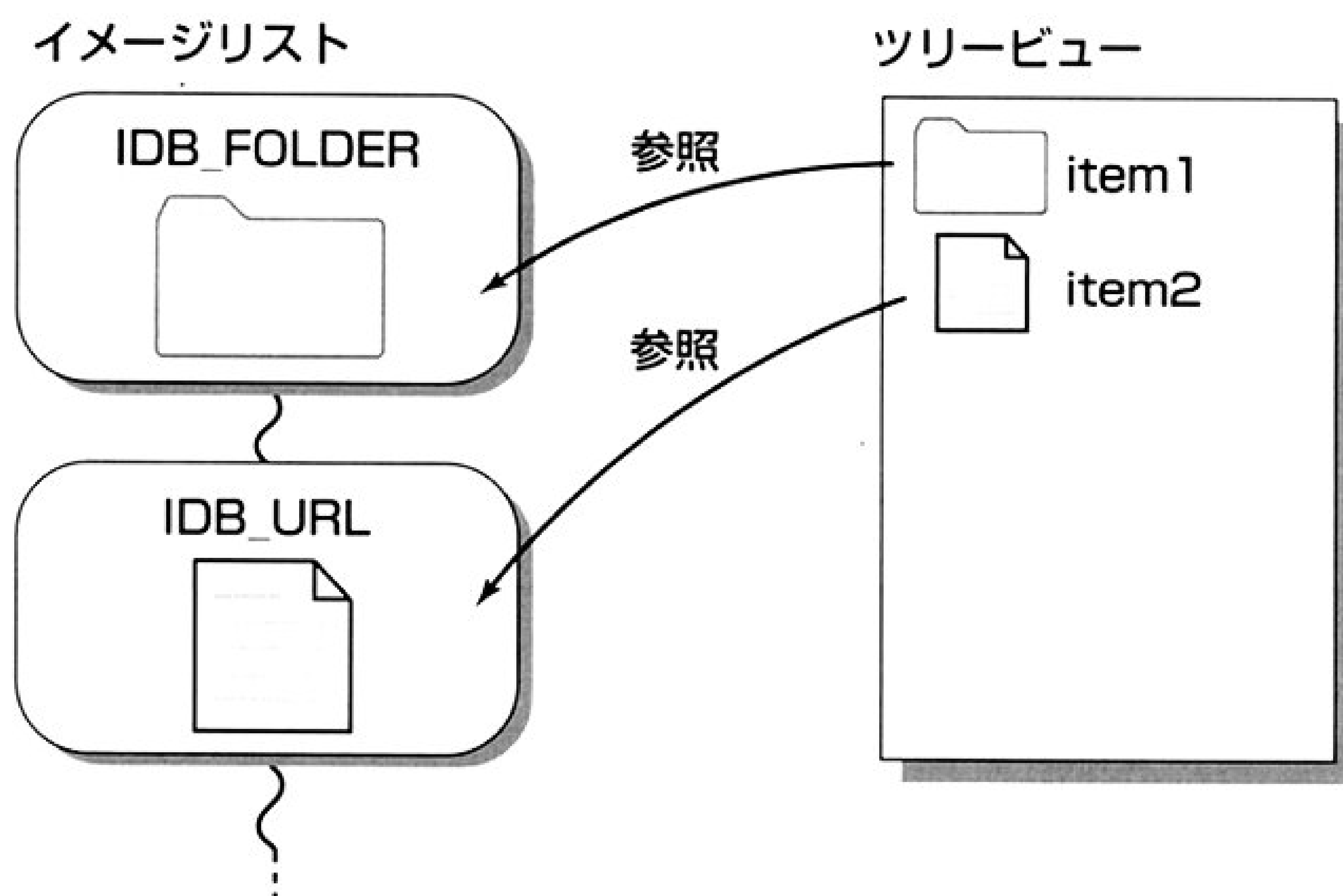


図 3-3　ツリービューがイメージリストを参照

CImageList クラスにはさまざまな機能がありますが、ツリービューコントロールと組み合わせて使う分には、「ビットマップをまとめて登録しておき、それらを配列を参照するように取り出して使う」と理解しておけばよいでしょう（**図 3-3**）。ツリービューコントロールにアイテムを挿入するときには、ラベルに使う文字列と同時にイメージリスト（ビットマップの配列）へのインデックスを指定します。すると、あとはツリービューコントロールが勝手に文字列と指定されたビットマップを表示してくれるのです。

それでは、上にあげた作業を順を追って見ていくことにしましょう。

●ビットマップの用意

URL マネージャでは 1 つのアイテムにつき、選択状態／非選択状態に対応する 2 つのビットマップを使います。URL マネージャでは URL 情報とフォルダの 2 種類のアイテムを使うので、合計 4 つのビットマップを用意します。4 つのビットマップには**図 3-4** に示すリソース ID を指定して、リソースエディタを使って作成します。

以上 4 つのビットマップリソースは付属 CD-ROM に用意してあるので、これをインポートして、URL マネージャのプロジェクトに組み込むことにしましょう。これは他のプロジェクトで作ったリソースを使い回すときに利用できるテクニックです。

リソースをインポートするためには、プロジェクトワークスペースからはアクセスでき

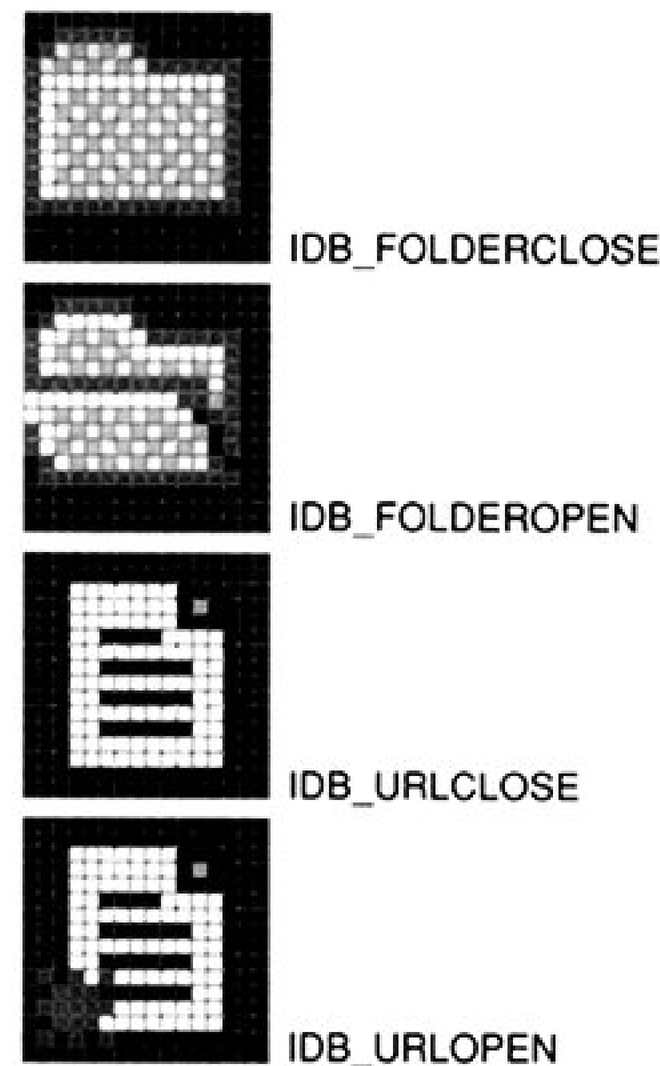


図3-4 作成するリソースとそのリソースID

ない、別のプロジェクトで管理されているファイルをアクセスしなければなりません。こうしたファイルにアクセスするためには、ファイルオープンダイアログボックスから必要なファイルを直接指定する必要があります。そのためには、プロジェクトの中でどのファイルがリソースを管理するために使われているのかを知る必要があります。第1部でプロジェクトを構成しているファイルの紹介を簡単にしましたが、そこで述べたようにリソースに関してはリソーススクリプトファイル（～.rc）に保存されています。AppWizardを使ってプロジェクトを生成した場合には、「<プロジェクト名>.rc」という名前のリソーススクリプトファイルが生成され、ここにプロジェクトで使用するリソースの定義が記述されるようになっています。たとえば、URLManという名前のプロジェクトだったら、URLMan.rcというリソーススクリプトファイルが生成されるわけです。

それではURLマネージャに必要なリソースを追加することにしましょう。付属CD-ROMで提供している「URLMan¥Resource」フォルダにはresource.rcというファイルが用意されています。このフォルダにはResourceという架空のプロジェクトがあるものとして、URLマネージャに必要なビットマップリソースがあらかじめすべて用意されています。ここからリソースをコピーすることによって、一気に必要なリソースを作ってしまいましょう（CD-ROMの中身はあらかじめハードディスクにコピーしておくことをお勧めします）。

まずメニューから［ファイル］－［開く］を実行して、先に述べたフォルダからResource.rcを開いてください。すると、ワークスペースウィンドウの［ResourceView］ページとよく似たウィンドウが開きます（**図3-5**）。

次にプロジェクトワークスペースに［ResourceView］ページを表示してください。あとの作業は簡単です。エクスプローラでファイルをコピーするように、Resource.rcのリソースウィンドウから、リソースを1つ1つドラッグしてプロジェクトワークスペースウィン

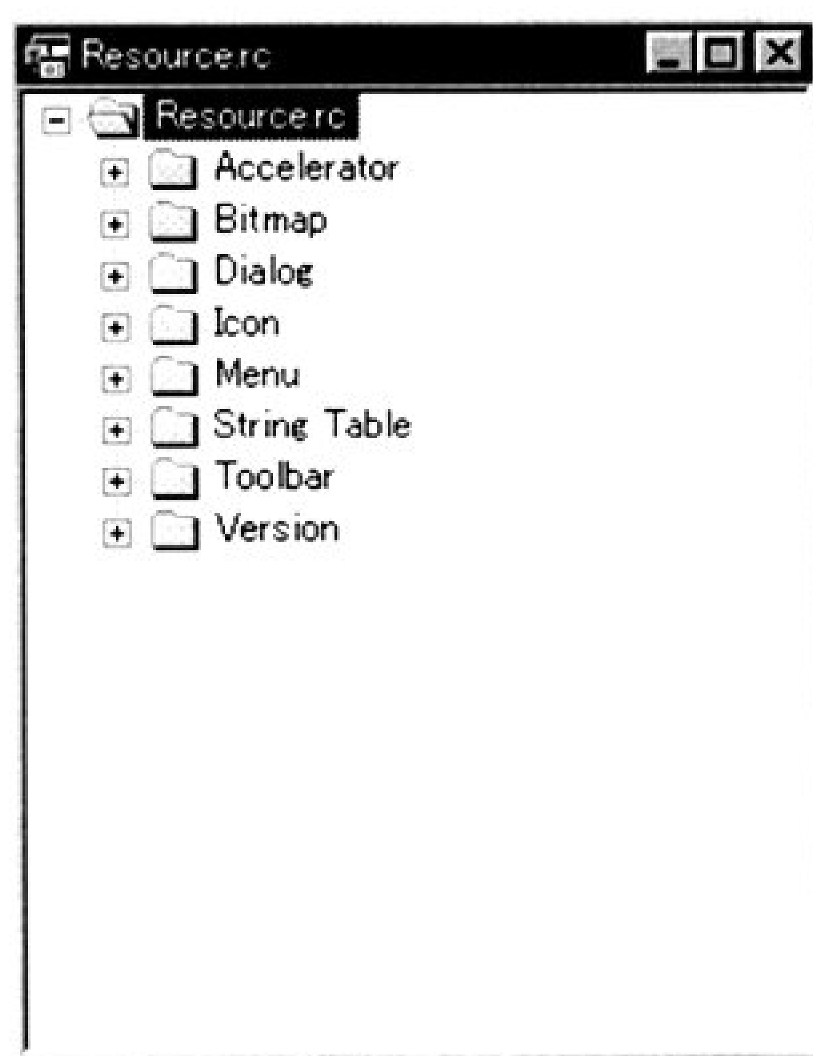


図 3-5 リソースウィンドウ

ドウにドロップするだけでリソースのコピーが完了します。このときドロップするときには Ctrl を押したままにしておいてください。離したままドロップするとコピーではなく移動になってしまいます。

Resource.rc ファイルに格納されている 4 つのビットマップリソースをコピーしたら、ビットマップの作成は終了です。

コピーが終了したら、シンボルブラウザで各ビットマップに割り当てられているリソース ID を確認してみてください。メニューの［表示］－［シンボルブラウザ］を実行すると、シンボルブラウザが表示されます（**図 3-6**）。

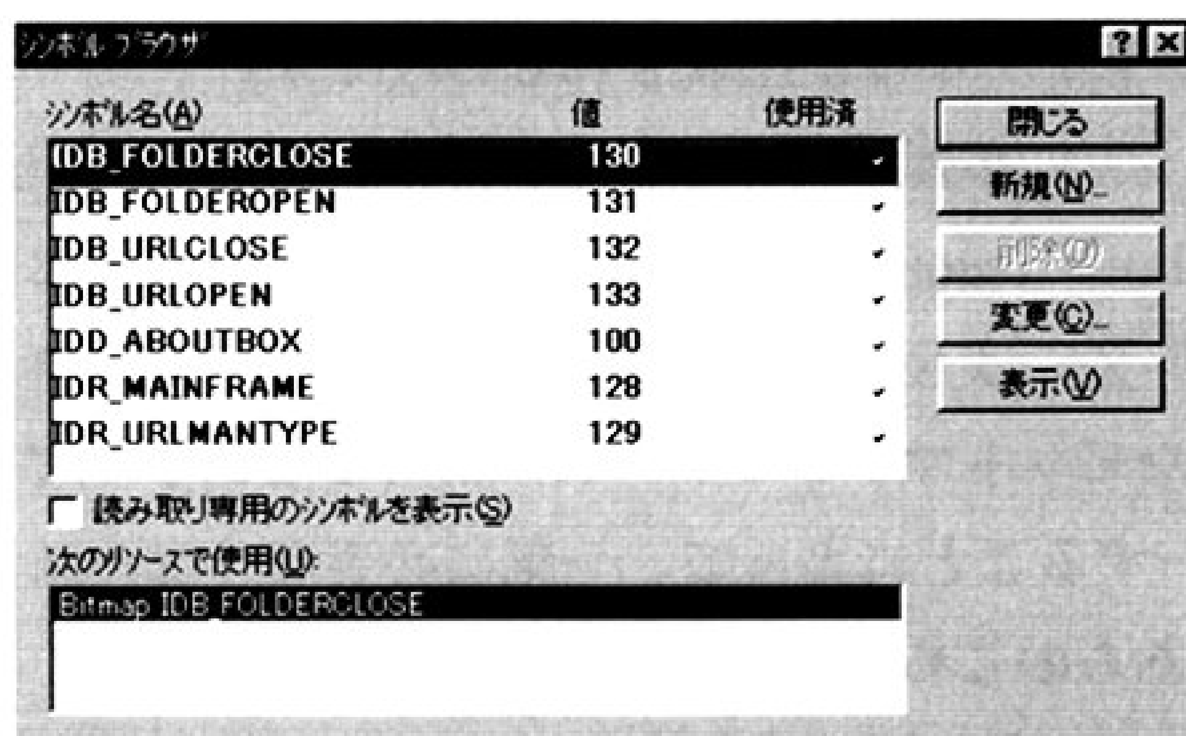


図 3-6 シンボルブラウザ

たった今コピーによって作成した IDB_FOLDERCLOSE に始まる 4 つのリソース ID を見ると、番号が連続していることがわかります。イメージリストに登録するビットマップリソースのリソース ID の値が連続していれば、ビットマップの登録を単純なループで処理できるため、意識的に番号付けを行っています。Developer Studio で作成したリソース

のリソースIDの値は、1つ作成するごとに1増加していくので、必要な数だけ先にビットマップを作成してリソースIDを確保し、それからデザインを行うとよいでしょう。

●ビットマップの登録

ツリービューコントロールで使うイメージリストの作成は、ツリービューコントロールにアイテムを挿入する前に行わなければならないので、通常はCView::OnInitialUpdate仮想メンバ関数をオーバーライドして、そこで行います。CView::OnInitialUpdate仮想メンバ関数は、CView::OnDrawメンバ関数が初めて呼び出される前に呼び出されるメンバ関数ですから、表示関係の初期化を行うのに適しています。ただし、SDIアプリケーションの場合には、［ファイル］－［新規作成］や［ファイル］－［開く］などを実行して、ドキュメントの初期化を行った場合にも呼び出されるので、この点を考慮する必要はあります。

CURLTreeView::OnInitialUpdate仮想メンバ関数もやはりAppWizardによってすでに作成されているので、**リスト3-2**に示すコードを書き加えてください。またCURLTreeViewクラスにCImageListクラスへのポインタを追加するのも忘れないでください（**リスト3-3**）。メンバ変数の追加は、手作業で行ってもかまいませんし、ワークスペースウィンドウを右クリックして、ショートカットメニューから［変数の追加］を選択する手段を使ってもかまいません。さらに、CURLTreeView::m_pImageListメンバ変数の初期化と後始末をするために、**リスト3-4**に示すように、CURLTreeViewクラスのコンストラクタとデストラクタにコードを追加してください。

リスト3-2　CURLTreeView::OnInitialUpdateメンバ関数（URLTreeView.cpp）

```
void CURLTreeView::OnInitialUpdate()
{
  CTreeCtrl& treeCtrl = GetTreeCtrl();

  if (m_pImageList == NULL) {
    CBitmap bitmap;
    m_pImageList = new CImageList;
    m_pImageList->Create(16, 16, TRUE, 4, 2);
    for (int i = 0; i < 4; i++) {
      bitmap.LoadBitmap(IDB_FOLDERCLOSE + i);
      m_pImageList->Add(&bitmap, RGB(0x80, 0x00, 0x00));
      bitmap.DeleteObject();
    }
    treeCtrl.SetImageList(m_pImageList, TVSIL_NORMAL);
  }

  CTreeView::OnInitialUpdate();
}
```

リスト3-3　メンバ変数の追加（URLTreeView.h）

```
class CURLTreeView : public CTreeView
{
...
public:
  CImageList* m_pImageList;
...
};
```

リスト3-4　m_pImageListメンバ変数の初期化と後始末（URLTreeView.cpp）

```
CURLTreeView::CURLTreeView()
{
  m_pImageList = NULL;
}

CURLTreeView::~CURLTreeView()
{
  delete m_pImageList;
}
```

では、CURLTreeView::OnInitialUpdate関数の中身を順を追って説明しましょう。

ツリーコントロールの取得

OnInitialUpdate関数では、最初に

```
CTreeCtrl& treeCtrl = GetTreeCtrl();
```

を実行して、ツリーコントロール（CTreeCtrlクラス）のオブジェクトを取得しています。**ツリーコントロール**はエディットボックスやリストボックスなどと同じコントロールの1種で、CTreeViewクラスのほとんどの機能を賄っています。実際CTreeViewクラスはメンバ関数をほとんど持たず（コンストラクタとGetTreeCtrlメンバ関数だけ）、CViewクラスとCTreeCtrlクラス（ツリーコントロール）を単純に組み合わせただけのクラスです。したがって、CTreeViewクラスの派生クラスでは、何をするにもまずはCTreeCtrlオブジェクトを取得することから始めるのです。

イメージリストの作成

ツリービューコントロールオブジェクトを取得したら、次にイメージリストを作成するために

```
m_pImageList = new CImageList
m_pImageList->Create(16, 16, TRUE, 4, 2);
```

という2行を実行します。イメージリストは、MFCの多くのクラスで見られるように、2つのステップを経て作成します。まず、コンストラクタを起動してCImageListオブジェクトを作成し、次にCImageList::Createメンバ関数を呼び出して、ビットマップを登録するための箱を用意します。これらCImageList::Createメンバ関数の書式は次のようになっています。

BOOL CImageList::Create(int *cx*, int *cy*, BOOL *bMask*, int *nInitial*, int *nGrow*)

返り値	**BOOL**	成功すればTRUE、失敗すればFALSE
引数	**int** *cx*	登録するビットマップの幅
	int *cy*	登録するビットマップの高さ
	BOOL *bMask*	ビットマップがマスクを持つならばTRUE
	int *nInitial*	初期状態での登録可能なビットマップの個数の上限
	int *nGrow*	上限値を超えてビットマップを登録したときに増加する登録領域の数

ここでは16×16のビットマップを4つ登録するので、「cx = cy = 16、nInitial = 4」としています。URLマネージャでは最初に4つのビットマップを登録するだけで、それ以上のビットマップを登録することはありませんが、仮にするとすれば2つのビットマップを組にして(選択時と非選択時のもの)登録することになるので、「nGrow = 2」としています。なお後述しますが、ビットマップにはマスクを使うので、「bMask = TRUE」としています。

ビットマップのイメージリストへの登録

イメージリストを作成したら、次にさきほど用意したビットマップをイメージリストに追加します。

```
for (int i = 0; i < 4; i++) {
  bitmap.LoadBitmap(IDB_FOLDERCLOSE + i);
  m_pImageList->Add(&bitmap, RGB(0x80, 0x00, 0x00));
  bitmap.DeleteObject();
}
```

ここでは4回ループを回して、4個のビットマップをイメージリストに登録しています。これには、まずCBitmap::LoadBitmapメンバ関数でビットマップリソースを1つbitmapオブジェクトに読み込んで、次にCImageList::Addメンバ関数で、読み込んだビットマップをイメージリストに登録し、最後にbitmapオブジェクトに読み込んだビットマップをCBitmap::DeleteObjectメンバ関数を呼び出して削除するという手順を踏みます。CImageList::Addメンバ関数の書式は次のようになっています。

int CImageList::Add(CBitmap* *pbmImage*, COLORREF *crMask*)

返り値	**int**	成功すればイメージリストへのインデックス(0から始まる)、失敗すれば-1
引数	**CBitmap*** *pbmImage*	登録するビットマップオブジェクトへのポインタ
	COLORREF *crMask*	マスクに使う色を指定

CImageList::Add メンバ関数の crMask 引数の役割は、登録するビットマップの中で使っている色から、透明色として扱う色を宣言することです。ビットマップは常に矩形ですから、これを描画すると背景によらず必ず矩形領域がビットマップで塗りつぶされてしまいます。たとえば10円硬貨のビットマップを作り、硬貨の中身だけを描画したいとしましょう。このとき硬貨の外側は必要ないのですが、ビットマップは必ず矩形ですから、とりあえず外側は暗赤で塗りつぶします。これを単純に描画すると外側の暗赤の部分も暗赤として描画されてしまうわけです。そこでイメージリストでは透明色を擬似的に作ることによって、硬貨の部分だけを描画できるようにしているのです。マスクを指定したビットマップを描画すると、ビットマップの中から crMask に指定した色の部分を抜かして描画されます。crMask に RGB (0x80, 0x00, 0x00) を指定しておけば、さっきの10円硬貨も余計な暗赤の外枠が描画されずに済むというわけです。

crMask に指定する色は、たまたま URL マネージャでは暗赤を使っていますが、ビットマップの中の描画されたい箇所では使われていない色ならば、何色でもかまいません。ビットマップの編集中はマスクに使う色が透明ではなく、本来の色で表示されてしまいますが、アプリケーションで描画するときにはちゃんと透明になります。

なおイメージリストにはアイコンを登録することもできますが、アイコンリソースにはもともと「透明色」を指定できるようになっているので、プログラム中でマスクの指定をする必要はありません。そのかわり、アイコンリソースを作る段階で忘れずに透明色を決めて作業しなければなりません。

イメージリストのツリーコントロールへの登録

イメージリストの作成が終わったら、

```
treeCtrl.SetImageList(m_pImageList, TVSIL_NORMAL);
```

として、CTreeCtrl::SetImageList メンバ関数を使ってツリーコントロールにイメージリストを登録します。CTreeCtrl::SetImageList メンバ関数は次のような関数です。

CImageList* CTreeCtrl::SetImageList(CImageList* *pImageList*, int *nImageListType*)

返り値	**CImageList***	ツリーコントロールに登録されていたイメージリスト（なければNULL）
引数	**CImageList*** *pImageList*	登録するイメージリスト
	int *nImageListType*	TVSIL_NORMAL、またはTVSIL_STATEを指定

nImageListTypeにはTVSIL_NORMALまたはTVSIL_STATEを指定できますが、選択／非選択の2つの状態に応じて表示するビットマップを切り替えるときには、TVSIL_NORMALを指定します。この場合は選択されているかどうかや、状態に応じて利用するビットマップなど、すべてをコントロールが自動的に処理してくれます。それ以外の使い方をするときには、TVSIL_STATEを指定して、アイテムの状態の管理やビットマップの使い分けについてのコードを追加する必要があります。たとえば、HTMLヘルプの左側のペインにあるツリービューコントロールでは、項目の選択／非選択ではなく、下の階層が展開されて見えるようになっているかどうかでビットマップの切り替えを行っています。

以上の作業でイメージリストの登録は終了です。これで、ツリービューコントロールにアイテムを登録するための準備が整いましたから、次節ではこれを行うためのコードについて解説をすることにしましょう。

3.4 ツリービューへのアイテムの挿入

URLマネージャでは、ツリービューに1つだけ「お気に入り」というアイテムを挿入した状態で起動します。そこで、ここでは「お気に入り」というラベルを持つアイテムを挿入する手順を例に取りながら、ツリービューへのアイテムの挿入方法を解説しましょう。

ツリービューにアイテムを挿入するには以下の手順を踏みます。

1. TV_INSERTSTRUCT構造体のオブジェクトに挿入するアイテムの情報を設定する
2. ツリービューへアイテムを挿入する

● TV_INSERTSTRUCT構造体

ツリーコントロールにアイテムを登録するときには、TV_INSERTSTRUCT構造体を使います。TV_INSERTSTRUCT構造体の中にはさらにTV_ITEM構造体が含まれていて、アイテムに関するパラメータはすべてこちらに含まれています。TV_ITEM構造体はアイテムの挿入以外にも多くの場面で使われるために、このような二重構造になっています。

```
typedef struct _TV_INSERTSTRUCT {
    HTREEITEM hParent;        // hParent と hInsertAfter で
    HTREEITEM hInsertAfter;   // 挿入位置を指定
    TV_ITEM   item;           // 挿入するアイテムの内容
} TV_INSERTSTRUCT;
```

ツリービューコントロールのアイテムはツリー状の階層構造を持っているので、アイテムを挿入する位置は、リストボックスのように先頭からのインデックスを指定するだけでは決まりません。そこで、インデックスの代わりに、hParent メンバ変数で親子関係を指定し、さらに hInsertAfter メンバ変数によって同じ親を持つアイテム（兄弟：Sibling）間での位置を指定します。

hParent に指定するハンドルは、あとで解説する CTreeCtrl::InsertItem メンバ関数（新規アイテムを挿入する）や CTreeCtrl::GetSelectedItem メンバ関数（選択されているアイテムを取得する）などを使って取得できます。指定する親がいない場合には（最上位階層にアイテムを挿入する場合）、hParent には TVI_ROOT または NULL を指定します（効果は同じ）。具体的には次のような感じでアイテムを挿入していくことになります。

1. 最上位階層にアイテム 1 を挿入する
2. 1. で取得したハンドルを hParent に指定して、アイテム 2 を挿入する
3. 必要に応じて、2. を繰り返す

または、次のような使い方も考えられます。

1. ユーザーがツリービューでアイテムを選択し、そこに新規にアイテムを追加する
2. 選択されているアイテムのハンドルを GetSelectedItem メンバ関数で取得する
3. そのハンドルを hParent に指定して、新規に追加されたアイテムを挿入する

hInsertAfter には、**表 3-3** に示す 3 種類の値を指定できます。このようにアイテムを挿入できる位置は 3 種類に限られ、任意の位置にアイテムを挿入することはできません。

値	効果
TVI_FIRST	兄弟の中で先頭に挿入
TVI_LAST	兄弟の中で末尾に挿入
TVI_SORT	アイテムのラベルでソート

表 3-3 挿入位置

実際に挿入するアイテムの情報は最後のメンバ変数、TV_ITEM 構造体のオブジェクトに設定します。TV_ITEM 構造体のメンバ変数は**表 3-4** に示すようなものです。この構造体には 10 個のメンバ変数がありますが、すべてのメンバ変数に値を指定する必要はありません。TV_ITEM 構造体の mask メンバ変数にマスクフラグを指定すると、指定したフラ

グに対応するメンバ変数だけが解釈され、残りのメンバ変数は無視されるからです。たとえば、文字列（ラベル）と選択時／非選択時のビットマップからなるアイテムを挿入するのならば、

```
mask = TVIF_TEXT | TVIF_IMAGE | TVIF_SELECTEDIMAGE;
```

とするわけです。

ところで TV_ITEM 構造体はアイテムの挿入以外にも、すでに挿入済みのアイテムの値を取り出すときにも使います。その関係上、挿入時には使われないメンバ変数も含まれています。**表 3-4** に挿入時と取得時に分けてメンバ変数に指定する値を解説しておきます。

メンバ変数	マスクフラグ	挿入時の目的	取得時の目的
HTREEITEM hItem	TVIF_HANDLE	使用せず	取得するアイテムのハンドル
UINT state	TVIF_STATE	現在の状態	左に同じ
UINT stateMask	TVIF_STATE	利用する状態	左に同じ
LPSTR pszText	TVIF_TEXT	ラベルに設定する文字列	ラベルを受け取るバッファ
int cchTextMax	TVIF_TEXT	使用せず	ラベルを受け取るバッファの長さ
int iImage	TVIF_IMAGE	非選択時のイメージ（イメージリストへのインデックス）	左に同じ
int iSelectedImage	TVIF_SELECTEDIMAGE	選択時のイメージ（イメージリストへのインデックス）	左に同じ
int cChildren	TVIF_CHILDREN	コールバックの指定	子アイテムの有無
LPARAM lParam	TVIF_PARAM	ユーザーが定義	左に同じ

表 3-4　TV_ITEM 構造体のメンバ変数

● アイテムを挿入する

TV_INSERTSTRUCT 構造体に値を設定したら、CTreeCtrl::InsertItem メンバ関数を使ってツリービューコントロールにアイテムを挿入します。CTreeCtrl::InsertItem メンバ関数の引数には、設定をした TV_INSERTSTRUCT 構造体へのポインタを指定します。CTreeCtrl::InsertItem メンバ関数の書式は次のようになっています。

HTREEITEM CTreeCtrl::InsertItem(LPTV_INSERTSTRUCT *lpInsertStruct*)

返り値	**HTREEITEM**	挿入したアイテムを示すハンドル
引数	**LPTV_INSERTSTRUCT** *lpInsertStruct*	挿入するアイテム

ツリービューコントロールにアイテムを挿入するには、以上のように、構造体に必要なパラメータを設定し、CTreeCtrl::InsertItem メンバ関数を呼び出す、というステップを踏みます。

●「お気に入り」アイテムを挿入してみる

それでは URL マネージャで使用しているコードをもとに具体的なアイテムの挿入方法を解説しましょう。まずはツリービューコントロールにアイテムを1つ挿入するためのメンバ関数を CURLTreeView クラスに追加します。このメンバ関数では、TV_INSERTSTRUCT 構造体の初期化と、アイテムの追加を行う CTreeCtrl::InsertItem メンバ関数の呼び出しを行います。

いつものように、ワークスペースウィンドウの[ClassView]から、次のようにして CURLTreeView::InsertItem メンバ関数を作成してください。InsertItem 関数の書式は次のとおりです。

HTREEITEM CURLTreeView::InsertItem(HTREEITEM *hParent*, **LPSTR** *pszText*, **BOOL** *bFolder*, **LPARAM** *pos*)

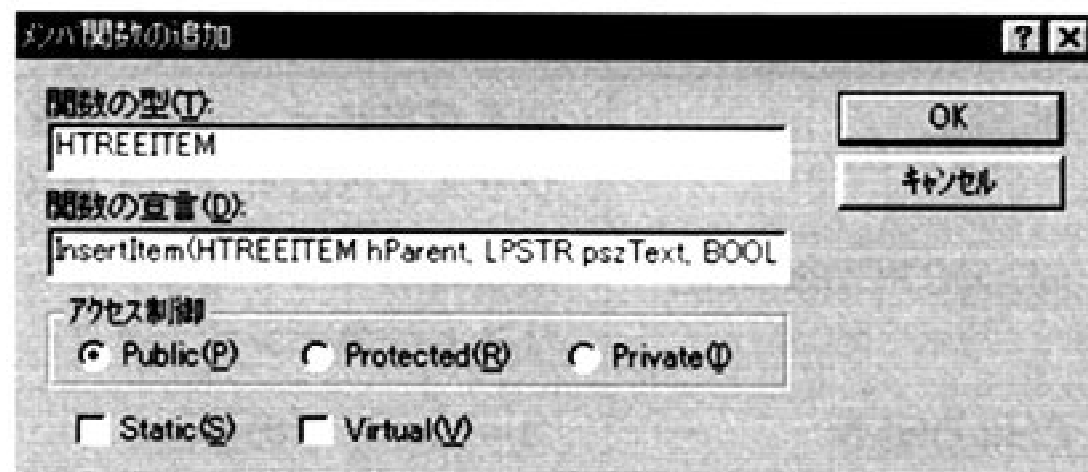


図 3-7　CURLTreeView::InsertItem メンバ関数の追加

さらに、たった今作成した CURLTreeView::InsertItem メンバ関数の中身に、**リスト 3-5** の内容を入力してください。

リスト 3-5　CURLTreeView::InsertItem メンバ関数（URLTreeView.cpp）

```
HTREEITEM CURLTreeView::InsertItem(
HTREEITEM hParent,  //親アイテムのハンドル
LPSTR pszText,      //ラベルに設定する文字列
BOOL bFolder,       //フォルダならば TRUE、URL 情報ならば FALSE
LPARAM pos)         //ドキュメントクラスに登録されている URL 情報へのインデックス
{
  TV_INSERTSTRUCT treeitem;
  CTreeCtrl& treeCtrl = GetTreeCtrl();
```

```
    treeitem.hParent = hParent;
    treeitem.hInsertAfter = TVI_SORT;
    treeitem.item.mask = TVIF_TEXT | TVIF_IMAGE |
                         TVIF_SELECTEDIMAGE | TVIF_PARAM;
    treeitem.item.pszText = pszText;
    treeitem.item.iImage = bFolder ? 0 : 2;
    treeitem.item.iSelectedImage = bFolder ? 1 : 3;
    treeitem.item.lParam = bFolder ? -1 : pos;

    HTREEITEM hItem = treeCtrl.InsertItem(&treeitem);

    return hItem;
}
```

では、CURLTreeView::InsertItemメンバ関数でTV_INSERTSTRUCT構造体およびTV_ITEM構造体に設定している値について解説しましょう。先にも解説したようにTV_ITEM構造体にはかならずしもすべてのメンバ変数に値を設定する必要はありません。ここで設定しているメンバ変数は**表3-5**に示す4つです。

メンバ変数	指定する値
pszText	ラベルに設定する文字列
iImage	非選択時のイメージ(フォルダならば0、URL情報ならば1)
iSelectedImage	選択時のイメージ(フォルダならば2、URL情報ならば3)
lParam	CURLManDoc::m_arrayURLメンバ変数へのインデックス。ツリービューに登録したアイテムに対応する情報を示す

表3-5　URLマネージャで使うTV_ITEM構造体のメンバ変数

これらのうちlParamメンバ変数については、その使い方はプログラマが自由に決めることができることになっています。つまりこのメンバ変数がツリービューコントロールの動作に影響を及ぼすことはありません。そこで、URLマネージャでは、ツリービューに登録したアイテムに対応する、ドキュメントクラスで管理されている情報を示すために使用しています。ドキュメントクラスではCURLManDoc::m_arrayURLメンバ配列変数に情報を格納していますから、この配列へのインデックスをlParamメンバ変数に保存しています。ただし、挿入するアイテムがフォルダであれば、そのことを示すためにlParamメンバ変数には-1を保存しています。この値の使い方はあとで解説しますが、ツリービューコントロールのアイテムをクリックしたときに、CBrowserViewクラスのペインに表示するWebページのURLを取得するために保存しています。とりあえず、今のところは保存するだけで参照されることはありません。

iImageメンバ変数とiSelectedImageメンバ変数には、CTreeCtrl::SetImageListメンバ関数で登録したイメージリストへのインデックスを指定しています。先に登録したイメージリストは

- インデックスが0：フォルダの非選択時のイメージ
- インデックスが1：フォルダの選択時のイメージ
- インデックスが2：URL情報の非選択時のイメージ
- インデックスが3：URL情報の選択時のイメージ

となっていましたから（**リスト3-2**）、そのように値を指定します。

以上、**表3-5**に示した4つのメンバ変数にだけ値を設定し、新たに挿入するアイテムのパラメータとするため、maskメンバ変数には次の値を設定しています。このように、利用するメンバ変数に対応するマスクフラグを論理和で加えた値を指定します。

```
treeitem.item.mask = TVIF_TEXT | TVIF_IMAGE |
                     TVIF_SELECTEDIMAGE | TVIF_PARAM;
```

TV_INSERTSTRUCT構造体のオブジェクト（treeitem）のメンバ変数に値を指定し終わったら、CTreeCtrl::InsertItemメンバ関数を呼び出してアイテムを挿入しますが、その使い方はすでに解説したとおりです。

これでツリーコントロールにアイテムを挿入する関数が完成したので、CFriendTreeView::OnInitialUpdateメンバ関数に、「お気に入り」アイテムを挿入するコードを追加します（**リスト3-6**）。さきほども述べたように、URLマネージャを起動すると、「お気に入り」アイテムがツリービューコントロールに挿入された状態でウィンドウが表示されますから、この作業はOnInitialUpdateメンバ関数で行うのが適切です。

リスト3-6 「お気に入り」アイテムを挿入する（URLTreeView.cpp）

```
void CURLTreeView::OnInitialUpdate()
{
  CTreeCtrl& treeCtrl = GetTreeCtrl();

  if (m_pImageList == NULL) {
    CBitmap bitmap;
    m_pImageList = new CImageList;
    m_pImageList->Create(16, 16, TRUE, 4, 2);
    for (int i = 0; i < 4; i++) {
      bitmap.LoadBitmap(IDB_FOLDERCLOSE + i);
      m_pImageList->Add(&bitmap, RGB(0x80, 0x00, 0x00));
      bitmap.DeleteObject();
    }
    treeCtrl.SetImageList(m_pImageList, TVSIL_NORMAL);
```

```
    }

    treeCtrl.DeleteAllItems();
    InsertItem(
      NULL,                    // 最上位階層にアイテムを挿入する
      _T("お気に入り"),        // ラベルに“お気に入り”を指定
      TRUE,                    // TRUE はフォルダを表す
      -1);                     // フォルダを挿入するので無条件に-1 を指定

    CTreeView::OnInitialUpdate();
}
```

ここでは、CURLTreeView::InsertItem メンバ関数を呼び出す前に、CTreeCtrl::DeleteAllItems メンバ関数を呼び出しています。このメンバ関数を呼び出すと、名前のとおりツリービューコントロールに登録されているアイテムをすべて削除して空にします。URL マネージャを起動した直後ならば、まだ何もアイテムは登録されていないので無意味ですが、データファイルを開いたときにもう一度「お気に入り」アイテムを挿入してしまったり、再度データファイルを開いたときに、前に表示していたデータが残ってしまうのを防ぐために呼び出しています。

BOOL CTreeCtrl::DeleteAllItems()

返り値　**BOOL**　　成功すればTRUE を、失敗すればFALSE を返す

リスト 3-6 では、ラベルに使う文字列に、単純に「"お気に入り"」を指定するのではなく、_T 関数を使っていますが、これはユニコードに対応するための措置の1つです。Visual C++では#define を使って_UNICODE が定義されていると、いくつかの関数がユニコード対応のものに置き変わるしくみが用意されています。_T 関数もそのうちの1つであり、通常は何も処理を行わずに引数の文字列をそのまま返すだけですが、_UNICODE が定義されてると引数の文字列をユニコードに変換して返すようになります。プログラム中に現れるリテラル文字列に_T 関数を使っておけば、わずかな変更を加えてコンパイルし直すだけでユニコードに対応したアプリケーションを作ることができます。

以上の作業で見た目の変化を確認できるだけのコードがそろったので、ひとまずビルドして、結果を確認しておきましょう。**図 3-8** のような画面が無事表示されたでしょうか。それでは続けて、読み込んだデータファイルをもとに、ツリービューコントロールへアイテムを挿入する部分の解説を行います。

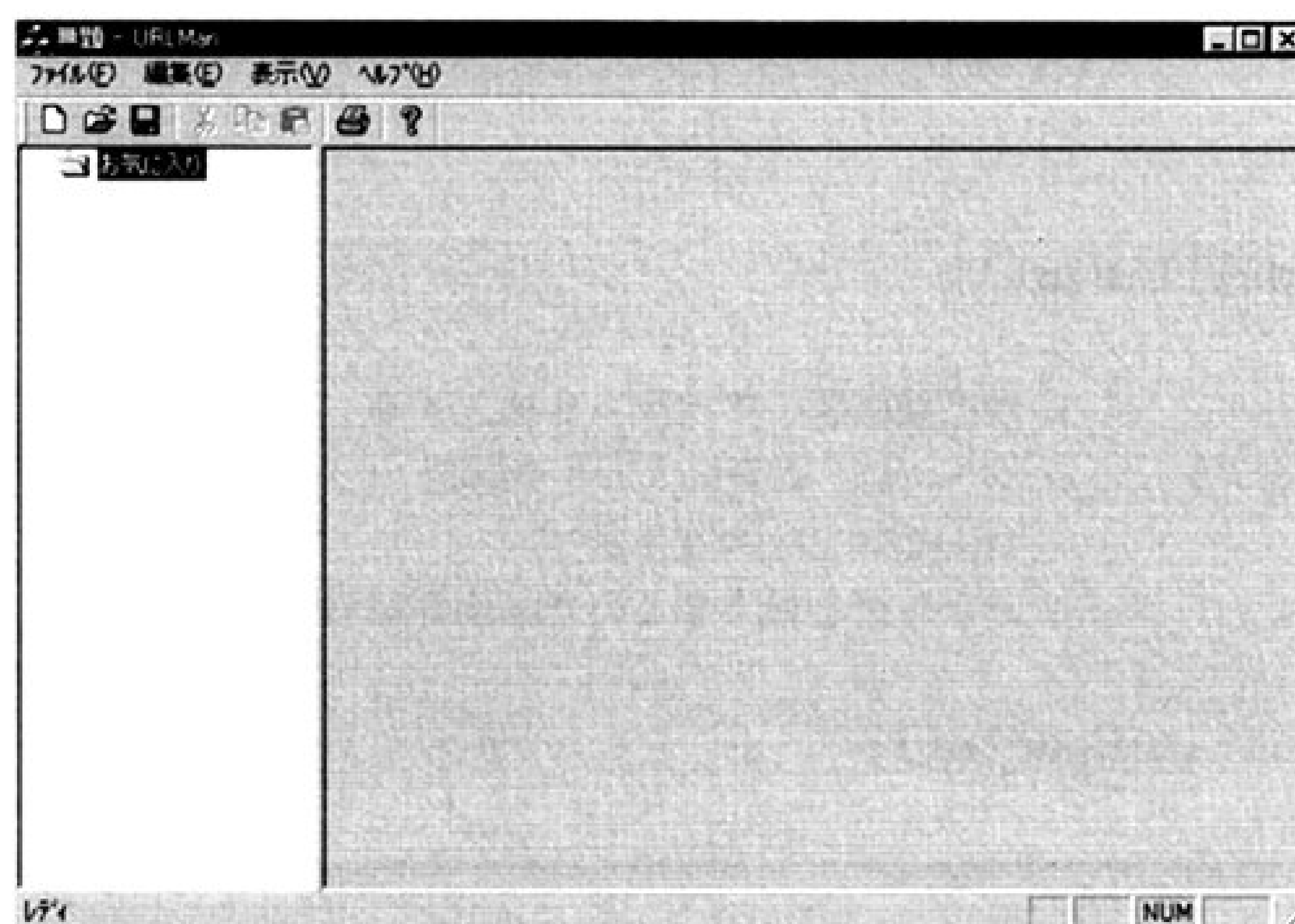


図3-8　ここまでの作業結果。まだデータファイルを読み込んでも表示されない

● データファイルをもとにアイテムを挿入する

「お気に入り」アイテムの挿入に続いて、読み込んだデータファイルをもとに、アイテムを挿入するコードを追加します。

この作業はCView::OnUpate仮想メンバ関数をオーバーライドして行います。この仮想メンバ関数は、CDocument::UpdateAllViewsメンバ関数を呼び出したときか、ビューが初期化されたときに呼び出されます。一般的には、ドキュメントが更新されたときに連動してビューを更新するために使われる仮想メンバ関数です。たとえば、ファイルをオープンすると、CDocumentクラスからCView::OnInitialUpdate仮想メンバ関数が呼び出され、そこからさらにCView::OnUpdate仮想メンバ関数が呼び出されます。ただし、CView::OnInitialUpdate仮想メンバ関数をオーバーライドしてしまうと、当然CView::OnUpdate仮想メンバ関数を呼び出すコードは上書きされ、呼び出されなくなってしまいます。そこで、CView::OnInitialUpdate仮想メンバ関数をオーバーライドするときには、明示的にCView::OnInitialUpdate仮想メンバ関数を呼び出すコードを含めておかなければなりません。このコードはすでに**リスト3-6**に含まれているので、「お気に入り」アイテムが挿入された直後にCURLTreeView::OnUpdateメンバ関数が呼び出されます。

CView::OnUpdateメンバ関数は次のように定義されています。

void CView::OnUpdate(CView* *pSender*, LPARAM *lHint*, CObject* *pHint*)

引数　**CView*** ***pSender***　OnUpdate仮想メンバ関数を呼び出すきっかけとなったビューオブジェクト。きっかけがドキュメントオブジェクトなどビューオブジェクト以外の場合はNULLとなり、すべてのビューの更新を意味する

LPARAM *lHint*　　用途は定められていない。ビューの更新に必要な情報の受け渡しに使う

CObject* *pHint*　　用途は定められていない。ビューの更新に必要な情報の受け渡しに使う

それでは、**表3-6**を参考にして、CURLTreeViewクラスでCView::OnUpdate仮想メンバ関数をオーバーライドしてください。この作業にはClassWizard、またはWizardBarを利用できます。

クラス	オブジェクトID	メッセージ
CURLTreeView	なし(CURLTreeViewを選択)	OnUpdate

表3-6　OnUpdate関数のオーバーライド

続いて、**リスト3-7**の内容を今作成したCURLTreeView::OnUpdate仮想メンバ関数に追加してください。

リスト3-7　CURLTreeView::OnUpdate仮想メンバ関数(URLTreeView.cpp)

```
void CURLTreeView::OnUpdate(CView* pSender, LPARAM lHint, CObject* pHint)
{
    if (pSender == NULL) {
        /* initialize tree */
        CURLManDoc* pDoc = GetDocument();
        CTreeCtrl& treectrl = GetTreeCtrl();

        HTREEITEM hTopItem = treectrl.GetFirstVisibleItem();
        // 先頭アイテムの位置を取得する
        HTREEITEM hCurrentItem = hTopItem;
        // アイテムを挿入する位置(親アイテムの位置)をhTopItemで初期化する
        CURLEntry* pURL = NULL;

        for (int i = 0; i < pDoc->m_nIndex; i++) {
            pURL = pDoc->m_arrayURL[i];
            if (pURL->getURL() == "dir") {
                // フォルダの作成
                // フォルダが作成されるのは、常に「お気に入り」の下
                hCurrentItem = InsertItem(
                    hTopItem,           // 挿入するアイテムの親アイテム
                    pURL->getTitle().GetBuffer(pURL->getTitle().GetLength()),
                                        // アイテムのラベルに設定する文字列
                    TRUE,               // TRUEならばフォルダのビットマップ
                    -1);                // フォルダのときは-1
                // URL情報アイテムを挿入する位置を表すhCurrentItemに
                // 今作ったフォルダの位置を保存する
```

```
                } else if (pURL->getURL() == "up") {
                    // フォルダの終了
                    hCurrentItem = hTopItem;
                    // アイテムが作成される親アイテムを hTopItem に戻す
                } else {
                    InsertItem(
                        hCurrentItem, // 挿入するアイテムの親アイテム
                        pURL->getTitle().GetBuffer(pURL->getTitle().GetLength()),
                                    // アイテムのラベルに設定する文字列
                        FALSE,      // FALSE ならば URL 情報のビットマップ
                        i);         // URL 情報の時は CURLManDoc::m_arrayURL へのインデックス
                }
            }
        }
    }
```

ここでのおおまかな作業は、データファイルから読み込んだデータが格納されているCURLManDoc::m_arrayURLメンバ配列変数を順にアクセスして、フォルダかURL情報かを判断しながら、適切なアイテムを挿入していくというものです。

それでは最初から順に見ていきましょう。先頭部分ではまずいつものようにCTreeCtrl::GetTreeCtrlメンバ関数を呼び出して、CTreeCtrl型オブジェクトを取得しています。このオブジェクトがなければツリービューコントロールを操作できないのはすでに述べたとおりです。

次に、CTreeCtrl::GetFirstVisibleItemメンバ関数を呼び出して、現在表示されている先頭のアイテムに対応するハンドルを取得しています。CURLTreeView::OnUpdate仮想メンバ関数が呼び出されるときには、CURLTreeView::OnInitialUpdate仮想メンバ関数で挿入した「お気に入り」アイテムだけが存在しているので、このアイテムが取り出されます。そして、このハンドルをhTopItem変数に保存しておきます。この変数はルートアイテムを表し、フォルダを作成する位置を示すために使われます。また、hTopItem変数をhCurrentItem変数にコピーしていますが、hCurrentItem変数はURL情報を格納するフォルダを示すために使われます。

そのあとのループでは、CURLManDoc::m_arrayURLメンバ配列変数の要素に1つ1つアクセスし、CURLEntry::m_strURLに格納されている文字列が“dir”ならばフォルダアイテムを挿入します。このとき、新たに作成したフォルダアイテムの位置を示すハンドル(CURLTreeView::InsertItemメンバ変数の返り値)をhCurrentItem変数に代入し、これ以後作成されるURL情報がこのフォルダの下に作成されるようにしています。

また“up”ならば、hCurrentItem変数へhTopItem変数の値を代入して、これ以後作成されるURL情報が挿入される位置をhTopItem変数に格納されている位置(「お気に入り」アイテムの下)に戻します。

それ以外ならば、URL情報としてアイテムを挿入しています。

このあたりの仕様についてはCURLManDocクラスの実装で述べたとおりです。

さて、これでやっとデータファイルを読み込んで、ツリービューコントロールに表示できるようになりました。さっそくビルドして、実行してみましょう。もう忘れてしまったかもしれませんが、URLマネージャで読み込むデータファイルを作成するには、lsurlツールを使います。

ただし、lsurlツールを使用するには、Developer StudioからURLMan¥lsURLフォルダにあるlsURLプロジェクトを開いて、プログラムをビルドする必要があります。

ビルドしたlsurl.exeはDOSプロンプトから次のようにして実行してください。

```
lsurl <出力ファイル名>

例:
lsurl favorites.lst
```

こうして作成したファイルをURLマネージャで開けば、図3-9のように表示されます。

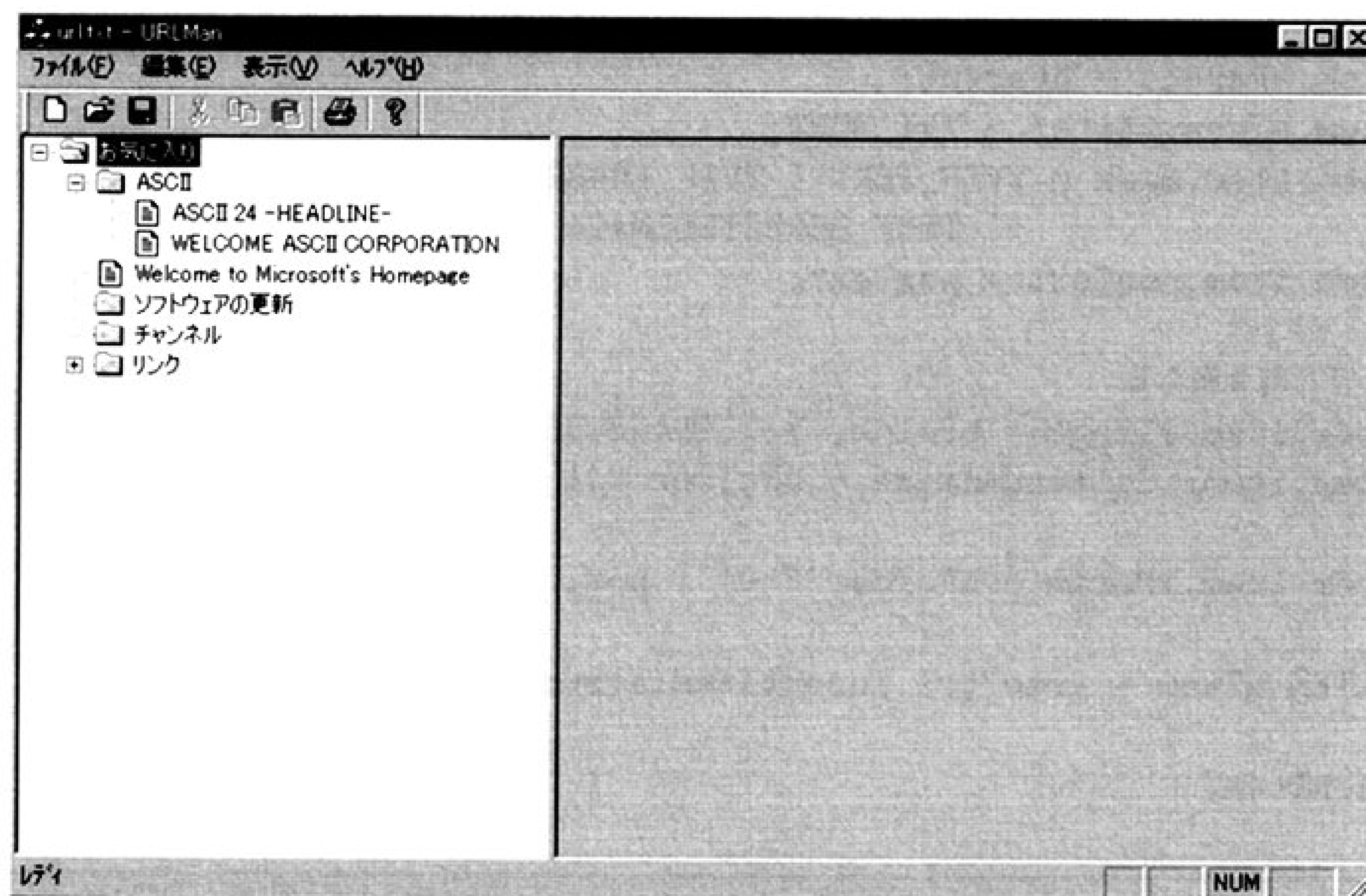


図3-9 ここまでの作業結果。ツリービューにアイテムが表示される

3.5 フォルダの状態によってビットマップを切り替える

URL マネージャも残すところ CBrowserView クラスの実装のみ、といきたいところですが、その前に1つだけ気になる点があるので、先に処理してしまいましょう。それは、フォルダのビットマップです。いじってみた方はわかると思いますが、フォルダも URL 情報も同じように、アイテムをクリックして選択されたときにビットマップが切り替わります。URL 情報はそれでもかまわないのですが、フォルダの方はやはりエクスプローラと同じように、下の階層が表示されているときにビットマップを切り替えたいところです。そこで、この処理を追加することにしましょう。

この処理を実現するためには、CURLTreeView::InsertItem メンバ関数を**リスト 3-8** のように2行変更します。

リスト 3-8　変更された CURLTreeView::InsertItem メンバ関数

```
HTREEITEM CURLTreeView::InsertItem(HTREEITEM hParent, LPSTR pszText,
                                    BOOL bDirectory, LPARAM pos)
{
  TV_INSERTSTRUCT treeitem;
  CTreeCtrl& treeCtrl = GetTreeCtrl();

  treeitem.hParent = hParent;
  treeitem.hInsertAfter = TVI_SORT;
  treeitem.item.mask = TVIF_TEXT | TVIF_IMAGE |
                        TVIF_SELECTEDIMAGE | TVIF_PARAM;
  treeitem.item.pszText = pszText;

  // 次の２行を書き換える
  treeitem.item.iImage = bFolder ? I_IMAGECALLBACK : 2;
  treeitem.item.iSelectedImage = bFolder ? I_IMAGECALLBACK : 3;

  treeitem.item.lParam = bFolder ? -1 : pos;

  HTREEITEM hItem = treeCtrl.InsertItem(&treeitem);

  return hItem;
}
```

挿入されるアイテムがフォルダのときだけ、TV_ITEM::iImage メンバ変数と TV_ITEM::iSelectedImage メンバ変数にイメージリストへのインデックスの代わりに、I_IMAGECALLBACK という特殊な値を代入しています。この値を代入しておくと、固定的にイメージリスト上のビットマップを指定することはなくなり、アイテムが表示されるときになると（アイテムが挿入されたり、クリックされたときなど）、TVN_GETDISPINFO 通知メッセー

ジがツリービューコントロールに送信されるようになります。そして、このメッセージハンドラで指定したイメージリストへのインデックスが使われるようになります。したがって、メッセージハンドラでフォルダが展開されているかを調べて、適切なインデックスを返せば、所望の機能が実現できるというわけです。

それでは、TVN_GETDISPINFO通知メッセージハンドラを作成しましょう。これにはClassWizardかWizardBarを使います。すると、CURLTreeView::OnGetdispinfoメンバ関数が作成されます。

クラス	オブジェクトID	メッセージ	メッセージハンドラ
CURLTreeView	なし（CURLTreeViewを指定）	TVN_GETDISPINFO	OnGetdispinfo

表3-7 TVN_GETDISPINFOメッセージハンドラ

続けて、**リスト3-9**に示すように、CURLTreeView::OnGetdispinfoメンバ関数の中身を入力してください。

リスト3-9 CURLTreeView::OnGetdispinfoメッセージハンドラ(URLTreeView.cpp)

```
void CURLTreeView::OnGetdispinfo(NMHDR* pNMHDR, LRESULT* pResult)
{
    TV_DISPINFO* pTVDispInfo = (TV_DISPINFO*)pNMHDR;

    HTREEITEM hItem = pTVDispInfo->item.hItem;
    CTreeCtrl& treectrl = GetTreeCtrl();

    if (pTVDispInfo->item.mask & TVIF_IMAGE ||
        pTVDispInfo->item.mask & TVIF_SELECTEDIMAGE) {
        if (pTVDispInfo->item.state & TVIS_EXPANDED) {
            pTVDispInfo->item.iImage =
                pTVDispInfo->item.iSelectedImage = 1;
        } else {
            pTVDispInfo->item.iImage =
                pTVDispInfo->item.iSelectedImage = 0;
        }
    }

    *pResult = 0;
}
```

ここで行っている処理は、アイテムが展開されているかどうかを調べ、展開されていればイメージリストへのインデックスとして1(開いているフォルダのビットマップ)を返し、そうでなければ0(閉じているフォルダのビットマップ)を返すというものです。

CURLTreeView::OnGetdispinfoメンバ関数は次のように定義されています。

void CURLTreeView::OnGetdispinfo(NMHDR* ***pNMHDR*, LRESULT*** ***pResult*)**

引数　**NMHDR*** ***pNMHDR***　TVN_GETDISPINFO メッセージの引き金となったアイテムの情報が格納される

　　　LRESULT* ***lResult***　返り値を lResult へ格納する

引数で渡される pNMHDR 変数は最初 NMHDR 構造体へのポインタとして渡されますが、実際には TV_DISPINFO 構造体へのポインタなので、キャストしてから利用します。TV_DISPINFO 構造体は次のように定義されています。必要な情報はすべて TV_ITEM 型 item メンバ変数の中に格納されています。TV_ITEM 構造体についてはすでに解説済みなので繰り返しは避けますが、すべてのメンバ変数にアクセスできるわけではないので、その点について解説します。TVN_GETDISPINFO メッセージハンドラで読み出せるメンバ変数は、mask、hItem、state、lParam の 4 つだけです。それ以外のメンバ変数には無意味な値が格納されているだけです。これら無意味な値が格納されているメンバ変数は、値を返すために使われます。ここではビットマップを切り替えるのが目的ですから、iImage メンバ変数と iSelectedImage メンバ変数へ値を格納して、メッセージハンドラを終了します。

リスト 3-10　TV_DISPINFO 構造体

```
struct TV_DISPINFO
{
    NMHDR    hdr;
    TV_ITEM  item;
  /*
  UINT      mask;       メッセージハンドラで返すべき値が指示されている
  HTREEITEM hItem;      メッセージハンドラで情報を設定するアイテム
  UINT      state;      hItem に対応するアイテムの状態
  UINT      stateMask;
  LPSTR     pszText;
  int       cchTextMax;
  int       iImage;
  int       iSelectedImage;
  int       cChildren;
  LPARAM    lParam;     用途は自由
  */
};
```

メッセージハンドラの中身を見てみましょう。まず最初の if 文では、TV_DISPINFO::item.mask メンバ変数を参照しています。TV_DISPINFO 構造体、および item.mask が取りうる値については**リスト 3-10** および**表 3-8** を参照してください。item.mask メンバ変数には、メッセージハンドラで返すべきメンバ変数はどれかを指示する値が格納されています。

ここでは、TVIF_IMAGE か TVIF_SELECTEDIMAGE だったときだけ、処理を行っています。なお、TVIF_TEXT や TVIF_CHILDREN を使うには、CTreeCtrl::InsertItem メンバ関数でアイテムを挿入するときに、I_IMAGECALLBACK と似た特殊な値を使う必要があります。ただし、ここではこれ以上触れません。

mask の値	目的
TVIF_TEXT	ラベル(pszText)を設定する
TVIF_IMAGE	非選択時のビットマップ(iImage)を設定する
TVIF_SELECTEDIMAGE	選択時のビットマップ(iSelectedImage)を設定する
TVIF_CHILDREN	子アイテムの数(cChildren)を設定する

表 3-8 mask メンバ変数が取りうる値

次に、TV_DISPINFO::item.state メンバ変数を参照して、アイテムが展開されているかどうかを判断しています。この変数の値は、**表 3-9** に示す値の論理和で構成されているため、単純に等号で比較するだけではなく、論理積を使って目的の値だけを取り出す必要があります。ここではアイテムが展開されているかどうかを調べたいので、TVIS_EXPANDED が含まれているかを調べています。そして、展開されていれば、iImage メンバ変数と iSelectedImage メンバ変数に 1 を、そうでなければ 0 を代入しています。URL マネージャでは選択されているか否かを無視して、展開されているかどうかだけでビットマップを切り替えるため、iImage メンバ変数と iSelectedImage メンバ変数には同じ値をセットしています。

state の値	意味
TVIS_BOLD	アイテムがボールド表示されている
TVIS_CUT	カット&ペースト処理のために選択されている
TVIS_DROPHILITED	ドラッグ&ドロップのターゲットとして選択されている
TVIS_EXPANDED	アイテムが展開され、子アイテムが表示されている
TVIS_EXPANDEDONCE	子アイテムが一度は展開されている
TVIS_EXPANDPARTIAL	子アイテムの一部が展開されている
TVIS_SELECTED	アイテムが選択されている

表 3-9 state メンバ変数が取りうる値

以上でビットマップの切り替えに関するコードの追加は終了です。ビルドして動作を確認してみてください。フォルダを選択してもビットマップは切り替わらず、展開したときだけ切り替われば成功です。

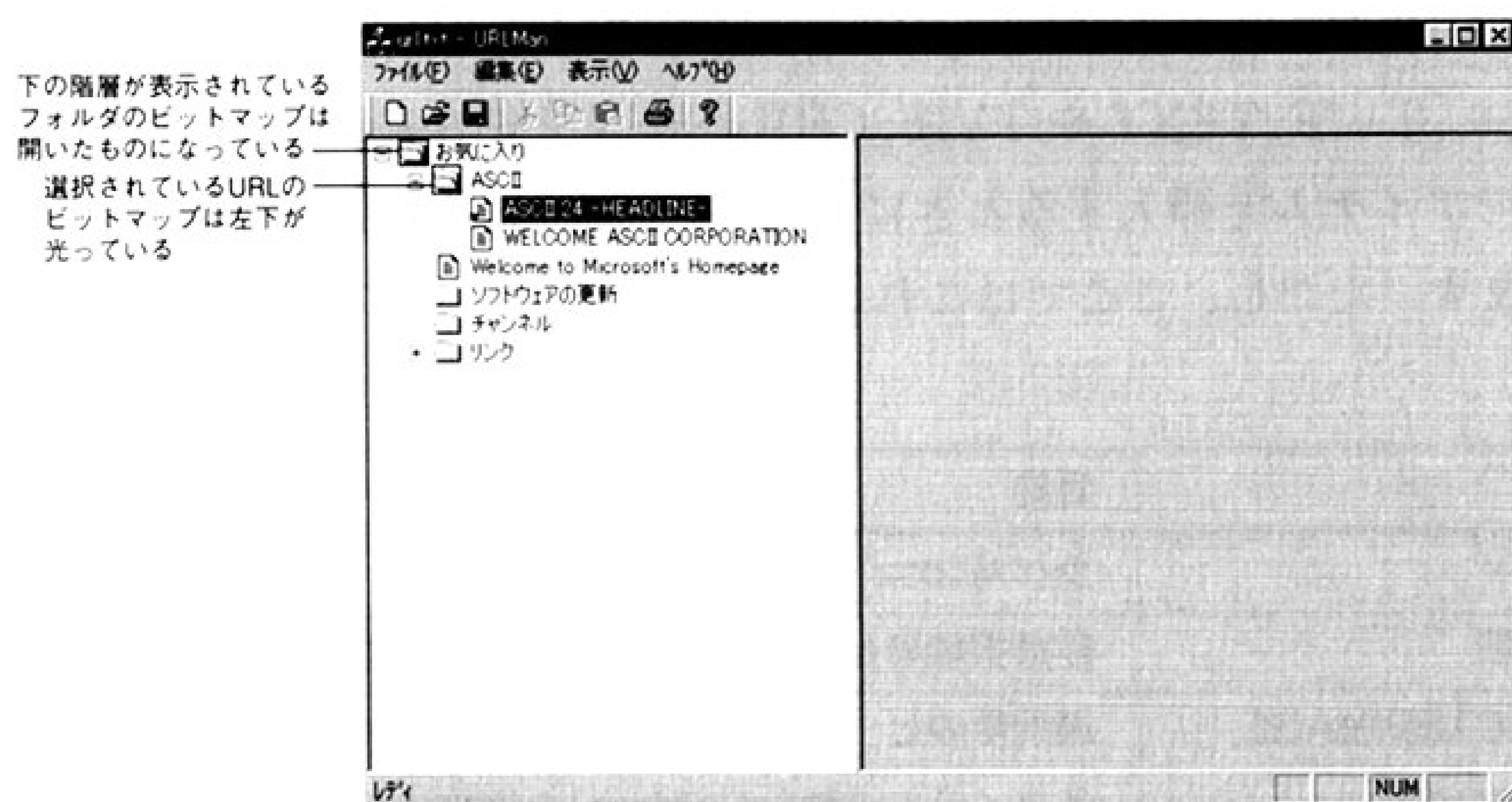


図 3-10 展開しているか否かでビットマップが切り替わるようになった

3.6 アイテムをクリックしてWebページを表示する

さあ、いよいよ大詰めです。最後に残された処理は、ツリービューコントロールのURL情報アイテムがクリックされたときに、HTMLコントロールにWebページを表示するコードを追加しましょう。この作業は、次のステップに分かれています。

1. ツリービューコントロールのアイテムがクリックされたことを検出する
2. ドキュメントオブジェクトから、クリックされたアイテムに対応するURLを取り出す
3. HTMLビューコントロールで、指定したURLを表示する

● アイテムがクリックされたことを検出する

Windowsプログラミングでは、ウィンドウに対するユーザーの操作はすべてメッセージという形でアプリケーションに通知されますが、ツリービューコントロールでもこれは例外ではありません。ツリービューコントロールに用意されている12種類の通知メッセージのなかからTVN_SELCHANGED通知メッセージをハンドルすれば、アイテムがクリックされて選択された瞬間を捕らえることができます。TVN_SELCHANGED通知メッセージは、選択されているアイテムが変更されたときに発行されるメッセージです。

それでは**表3-10**に従って、ClassWizardを使ってTVN_SELCHANGEDメッセージのハンドラを作成してください。

TVN_SELCHANGEDメッセージに対応するハンドラの書式は次のように定義されています。

クラス	オブジェクトID	メッセージ
CURLTreeView	なし(CURLTreeViewクラスを選択)	TVN_SELCHANGED

表3-10 TVN_SELCHANGEDメッセージのハンドラの指定

void CURLTreeView::OnSelchanged(NMHDR* *pNMHDR*, LRESULT* *pResult*)

引数 **NMHDR*** *pNMHDR* アイテムを選択したときの情報
LRESULT* *pResult* メッセージハンドラの結果を返す

pNMHDRはNMHDR構造体へのポインタとして渡されますが、実際にはNM_TREEVIEW構造体へのポインタとなっているので、キャストして利用します。NM_TREEVIEW構造体は次のようなメンバ変数を持っています。

```
typedef struct _NM_TREEVIEW {
    NMHDR     hdr;
    UINT      action;     // TVC_BYKEYBOARD or TVC_BYMOUSE or TVC_UNKNOWN
    TV_ITEM   itemOld;    // 直前に選択されていたアイテム
                             (hItem, state, lParam 以外の値は不定)
    TV_ITEM   itemNew;    // 新しく選択されたアイテム(同上)
    POINT     ptDrag;
} NM_TREEVIEW;
```

通知メッセージのメッセージハンドラには返り値がありませんが、代わりに*pResultに結果を代入してメッセージハンドラを終了することになっています。代入すべき値はマニュアルから通知メッセージを検索し(ここではTVN_SELCHANGED)、[Return Value]の項目に指示されている値を代入します。TVN_SELCHANGED通知メッセージの場合には「返り値は不要」とありますから、0でも代入しておけばよいでしょう。

URLマネージャでは**リスト3-11**のようにTVN_SELCHANGEDメッセージハンドラを実装しています。

リスト3-11 TVN_SELCHANGEDメッセージのハンドラ(URLTreeView.cpp)

```
void CURLTreeView::OnSelchanged(NMHDR* pNMHDR, LRESULT* pResult)
{
  NM_TREEVIEW* pNMTreeView = (NM_TREEVIEW*)pNMHDR;

  CURLManDoc* pDoc = GetDocument();

  CTreeCtrl& treectrl = GetTreeCtrl();

  TV_ITEM treeitem;
  treeitem.hItem = pNMTreeView->itemNew.hItem;
```

```
    treeitem.mask = TVIF_PARAM;
    treectrl.GetItem(&treeitem);

    if (treeitem.lParam != -1) {
      // クリックしたアイテムが URL 情報の場合
      pDoc->UpdateAllViews(this, treeitem.lParam, NULL);
      // CBrowserView::OnUpdate メンバ関数の呼び出しにつながる
    }

    *pResult = 0;
  }
```

ここでの処理の中心は、CTreeCtrl::GetItem メンバ関数の呼び出しと、CDocument::UpdateAllViews メンバ関数の呼び出しにあります。

まず CTreeCtrl::GetItem メンバ関数の呼び出しですが、これは選択されたアイテムに保存されている TV_ITEM::lParam メンバ変数の値を取り出すための処理です。アイテムを挿入するとき、アイテムが URL 情報ならば TV_ITEM::lParam メンバ変数にドキュメントオブジェクトへのインデックスを、フォルダならば-1 を保存しておいたことを覚えているでしょうか。この値を取り出せば、アイテムに対応する URL をドキュメントオブジェクトから取得できます。そこで、CTreeCtrl::GetItem メンバ関数によって lParam を取り出し、次の CDocument::UpdateAllViews メンバ関数の呼び出しで、CBrowserView クラスへと lParam の値を渡しているわけです。

CTreeCtrl::GetItem メンバ関数は次のように定義されています。

BOOL CTreeCtrl::GetItem(TV_ITEM* ***pItem*****);**

返り値	**BOOL**	正常終了した場合は TRUE、異常終了した場合は FALSE を返す
引数	**TV_ITEM*** *pItem*	情報を取得したいアイテムと情報種を指定する。また取得した情報が格納される。

CTreeCtrl::GetItem メンバ関数の使い方は、アイテムの挿入に使った CTreeCtrl::InsertItem メンバ関数とよく似ています。TV_ITEM 構造体を使うところは同じですし、値を設定したメンバ変数を示すためにマスクフラグを使うところも同じです。TV_ITEM 構造体やマスクフラグについては**表 3-5** を参照してください。ここでは、取得したい情報は lParam メンバ変数だけなので、TV_ITEM::mask メンバ変数に TVIF_PARAM だけを代入しています。また、TV_ITEM::hItem メンバ変数は取得するアイテムを示すために使われるので、マスクフラグにかかわらず必ず指定する必要があります。ここでは、前述した

NM_TREEVIEW構造体のメンバ変数から、新しく選択されたアイテムのハンドルを代入しています。

以上の準備をしてからCTreeCtrl::GetItemメンバ関数を呼び出すと、引数として渡したTV_ITEM構造体のメンバ変数が書き換えられ、情報を取得できます。ここでは、treeitem.lParamメンバ変数に、指定したアイテムが持つlParamの値が格納されてきます。

こうして取得したlParamの値をWebページを表示するために、CBrowserViewクラスへと渡す作業を行っているのがCDocument::UpdateAllViewsメンバ関数です。このメンバ関数はすでに解説したCView::OnUpdate仮想メンバ関数と組になっている関数であり、CDocument::UpdateAllViewsメンバ関数を呼び出すと、ドキュメントオブジェクトに対応しているすべてのビューのCView::OnUpdate仮想メンバ関数が呼び出されます。

● Webページを表示する

そこで、ここまで沈黙を保ってきたCBrowserViewクラスへとついに手を入れることにします。**表3-11**に従い、ClassWizardを使って、CView::OnUpdate仮想メンバ関数をオーバーライドして、CBrowserView::OnUpdate仮想メンバ関数を作成してください。ここでlParamの値を受け取り、Webページを表示する処理を行います。

クラス	オブジェクトID	メッセージ
CBrowserView	なし(CBrowserViewを選択)	OnUpdate

表3-11　CBrowserView::OnUpdate仮想メンバ関数

ここでの処理は、先のCDocument::UpdateAllViewsメンバ関数の引数に指定したlParamの値を受け取り、Webページを表示するという、ごく簡単なものです。**リスト3-12**の内容をCBrowserView::OnUpdate仮想メンバ関数の中に入力してください。

リスト3-12　CBrowserView::OnUpdate仮想メンバ関数(BrowserView.cpp)

```
void CBrowserView::OnUpdate(CView* pSender, LPARAM lHint, CObject* pHint)
{
    if (pSender != NULL) {
        CURLManDoc* pDoc = (CURLManDoc*)GetDocument();

        Navigate2(pDoc->m_arrayURL[lHint]->getURL());
    }
}
```

また、BrowserView.cppファイルの先頭部分に、次の1行を加えて、CURLManDocクラスにアクセスできるようにしてください。

リスト3-13　CURLManDocクラスにアクセスできるようにする(BrowserView.cpp)

```
#include "URLManDoc.h"
```

先にCDocument::UpdateAllViewsメンバ関数とCView::OnUpdateメンバ関数は組になっていると述べましたが、これには2つの意味があります。1つは、CDocument::UpdateAllViewsメンバ関数を呼び出すと、フレームワークからCView::OnUpdateメンバ関数が呼び出される点。もう1つは、CDocument::UpdateAllViewsメンバ関数の引数に指定した値が、そのままCView::OnUpdateメンバ関数の呼び出しに使われる点です。したがって、CBrowserView::OnUpdate仮想メンバ関数の引数lHintには、ツリービューコントロールで選択したアイテムのlParamの値(しつこいようですが、CURLManDoc::m_arrayURLメンバ配列変数へのインデックスです)が渡されます(CURLTreeView::OnSelchangedメンバ関数のCDocument::UpdateAllViewsメンバ関数を呼び出している箇所を参照のこと)。

この値さえ手に入れれば、あとの処理は単純です。CURLManDoc::m_arrayURL[lHint]を参照して、CURLEntry::getURLメンバ関数を呼び出し、ツリービューコントロールで選択したアイテムのURLを入手します。そして、CHtmlView::Navigate2メンバ関数を呼び出して、Htmlビューコントロールに Webページを表示します。

void CHtmlView::Navigate2(LPCTSTR *lpszURL*)

引数　**LPCTSTR** *lpszURL*　表示するWebページのURLを指定する

最後に、CBrowserView::OnInitialUpdate仮想メンバ関数を作成して、URLマネージャが起動した直後には、右側のペインにブランクページが表示されるようにしておきましょう。ClassWizardを使って、**表3-12**のようにCBrowserView::OnInitialUpdate仮想メンバ関数を作成してください。

クラス	オブジェクトID	メッセージ
CBrowserView	なし(CBrowserViewを選択)	OnInitialUpdate

表3-12　OnInitialUpdate関数のオーバーライド

すると、ClassWizardは初期状態でMicrosoftのWWWサイトを開くためのコードを追加します。Visual C++プログラマには有益な情報が満載されたWWWサイトではありますが、URLマネージャには不適当ですから、**リスト3-14**のように変更してしまってください。これで、起動直後にはブランクページが表示されるようになります。

リスト 3-14　CBrowserView::OnInitialUpdate 仮想メンバ関数

```
void CBrowserView::OnInitialUpdate()
{
    Navigator2(_T("about:blank"), NULL, NULL);
}
```

さあ！ これで URL マネージャはほぼ完成です。ビルドして動作を確認してみましょう。ツリービューコントロールでアイテムをクリックすると、Web ページが表示されるハズですが、どうでしょうか。



図 3-11　URL マネージャ、一応完成の図

4　もう少しWWWブラウザらしく

前章までで最低限ブラウズできるだけの作業を終えましたが、これでWWWブラウザを名乗るのは心苦しいものがありますから、もう少し周辺の環境を整えることにしましょう。

ここでは2つの機能をCBrowserViewクラスに追加します。1つは現在表示しているWebページのURLをキャプションに表示する機能です。Windowsアプリケーションでは、現在編集中のドキュメント名などをアプリケーション名と共にキャプションに表示するのが一般的な作法とされていますから、ぜひ実現しておきましょう。もう1つは、[先に進む]や[前に戻る]などのナビゲーション機能です。これはわざわざ作らなくとも、CBrowserViewクラスが管理しているウィンドウでは、右クリックすると表示されるコンテキストメニューから操作できますが、キーボードから操作できませんし、[停止]はこのコンテキストメニューにはありません。そこで、メニューからこれらの操作をできるようにしておきます。

4.1　キャプションにURLを表示する

キャプションに現在表示されているURLを表示するためには、

1. 表示が完了したことを検出し
2. 表示されているURLを調べて
3. キャプションへ適切な文字列を設定する

という3ステップが必要になります。それでは、順番に見ていくことにしましょう。

まずはWebページの表示完了を検出する手段ですが、この目的のためにCHtmlViewクラスには、CHtmlView::OnNavigateComplete2仮想メンバ関数が用意されています。この仮想メンバ関数は標準では何も処理を行いませんが、Webページの表示が完了すると、フレームワークから呼び出されます。つまり、MFCで定義されている、ただオーバーライドされるのを待っているだけの仮想メンバ関数の1つというわけです。

CHtmlViewクラスにはこうした状態の変化を捕らえるための仮想メンバ関数がいくつも用意されています。ここでは利用しませんが、**表4-1**に簡単に紹介しておきます。

仮想メンバ関数	目的
OnBeforeNavigate2	ナビゲーション開始前に呼び出される
OnStatusTextChange	ステータスバーのテキストが変更されたことを通知するために呼び出される
OnProgressChange	ダウンロード操作の進行状況が変更されたことを通知するために呼び出される。進行状況を表すプログレスバーを実装するために利用する
OnCommandStateChange	[次に進む]、[前に戻る]などのコマンドの有効状態が変化したことを通知するために呼び出される
OnDownloadBegin	ダウンロード開始前に呼び出される。ただし短時間でダウンロードが終了した場合には呼び出されない
OnDownloadComplete	ダウンロードの成功／失敗にかかわらず、ナビゲーションが終了したことを通知するために呼び出される
OnDocumentComplete	OnDownloadBeginに対応して、ダウンロードの終了を通知するために呼び出される。必ずしも呼び出されるとは限らない

表4-1 CHtmlViewクラスの代表的なメンバ関数

では、CHtmlView::OnNavigateComplete2仮想メンバ関数の解説を続けます。この仮想メンバ関数は次のように定義されています。

void CHtmlView::OnNavigateComplete2(LPCTSTR *strURL*)

引数 **LPCTSTR** *strURL* 表示を完了したURLが格納される

したがって、この仮想メンバ関数をオーバーライドすれば、自動的に現在表示されているURLも入手できるというわけです。それでは、ClassWizardを使って、CHtmlView::OnNavigateComplete2仮想メンバ関数をオーバーライドしてください。

クラス	オブジェクトID	メッセージ
CBrowserView	なし(CBrowserViewを選択)	OnNavigateComplete2

表4-2 OnNavigateCompleteメンバ関数のオーバーライド

次にウィンドウのキャプションへ文字列を設定する方法ですが、これにはCWnd::SetWindowTextメンバ関数を使います。

void CWnd::SetWindowText(LPCTSTR *lpszString*)

引数 **LPCTSTR** *lpszString* キャプションへ設定する文字列を指定する

キャプションを設定するための関数のわりには、SetWindowTextなどという曖昧な名前付けがされているのには理由があります。実はCWnd::SetWindowTextメンバ関数は必ずしもキャプションを設定するとは限らないのです。たとえば、エディットボックスコントロールに対応しているCEditクラスはCWndクラスの派生クラスですから、CEdit::SetWindowTextメンバ関数が定義されていますが、CEdit::SetWindowTextメンバ関数の処理は、エディットボックスに文字列を入力するというものです。また、スタティックコントロールに対応するCStaticクラスのCStatic::SetWindowTextメンバ関数の処理は、ラベル文字列を設定するというものです。これらコントロール(コントロールはすべてウィンドウです)やビューウィンドウを見てもわかるように、ウィンドウには必ずしもキャプションがあるとは限りません。実際、URLManプロジェクトには数々のウィンドウが含まれていますが、キャプションを持っているのはCMainFrameクラスに対応するメインフレームウィンドウだけです。こうしたウィンドウでは、キャプション以外の文字列を操作するためにSetWindowTextメンバ関数が使われるため、こんな名前になっているのです。

以上でキャプションにURLを表示するための材料はそろいましたから、**リスト4-1**を見ながら、先に作成しておいたCBrowserView::OnNavigateComplete2仮想メンバ関数を実装してください。

リスト4-1　CBrowserView::OnNavigateComplete2仮想メンバ関数(BrowserView.cpp)

```
void CBrowserView::OnNavigateComplete2(LPCTSTR strURL)
{
  CString title;
  CString url(strURL);

  title = url + " - " + AfxGetAppName();

  CFrameWnd* pWnd = GetParentFrame();
  pWnd->SetWindowText(title);

  CHtmlView::OnNavigateComplete2(strURL);
}
```

まず、キャプションに設定する文字列をtitle変数を使って作ります。Windowsアプリケーションでは、キャプションに「<編集中のドキュメント名> - <アプリケーション名>」と設定するのがスタイルですから、引数として渡されたstrURL変数と、AfxGetAppName関数で取得できるアプリケーション名を連結して、文字列を作ります。

次に、メインフレームウィンドウのキャプションを変更するために、メインフレームウィンドウオブジェクトへのポインタをCWnd::GetParentFrameメンバ関数を使って取得します。親ウィンドウへのポインタを取得するためのメンバ関数としては、CWnd::GetParentメンバ関数がありますが、こちらはフレームウィンドウであろうとなかろうと、とにかく

親ウィンドウを返します。それに対して、CWnd::GetParentFrame メンバ関数は、親ウィンドウがフレームウィンドウでなければ、そのまた親ウィンドウを取得して、フレームウィンドウが見つかるまで繰り返します。実のところ AppWizard が生成するスケルトンの場合、ビューウィンドウはフレームウィンドウの子ウィンドウなので CWnd::GetParent メンバ関数を使っても同じ結果が得られるのですが、URLMan の場合スプリットウィンドウを使用しているため、CBrowserView オブジェクトのウィンドウの親ウィンドウは、メインフレームウィンドウではなく、スプリットウィンドウになっています。そこで、ここでは CWnd::GetParentFrame メンバ関数を使います。

最後に、CWnd::GetParentFrame メンバ関数で取得したオブジェクトポインタを使って、CWnd::SetWindowText メンバ関数を呼び出し、キャプションの設定は終了です。

ひとまずここまでの作業を確認するため、ビルドして実行してみましょう。**図 4-1** のようにキャプションに現在表示されている URL が表示されたら成功です。

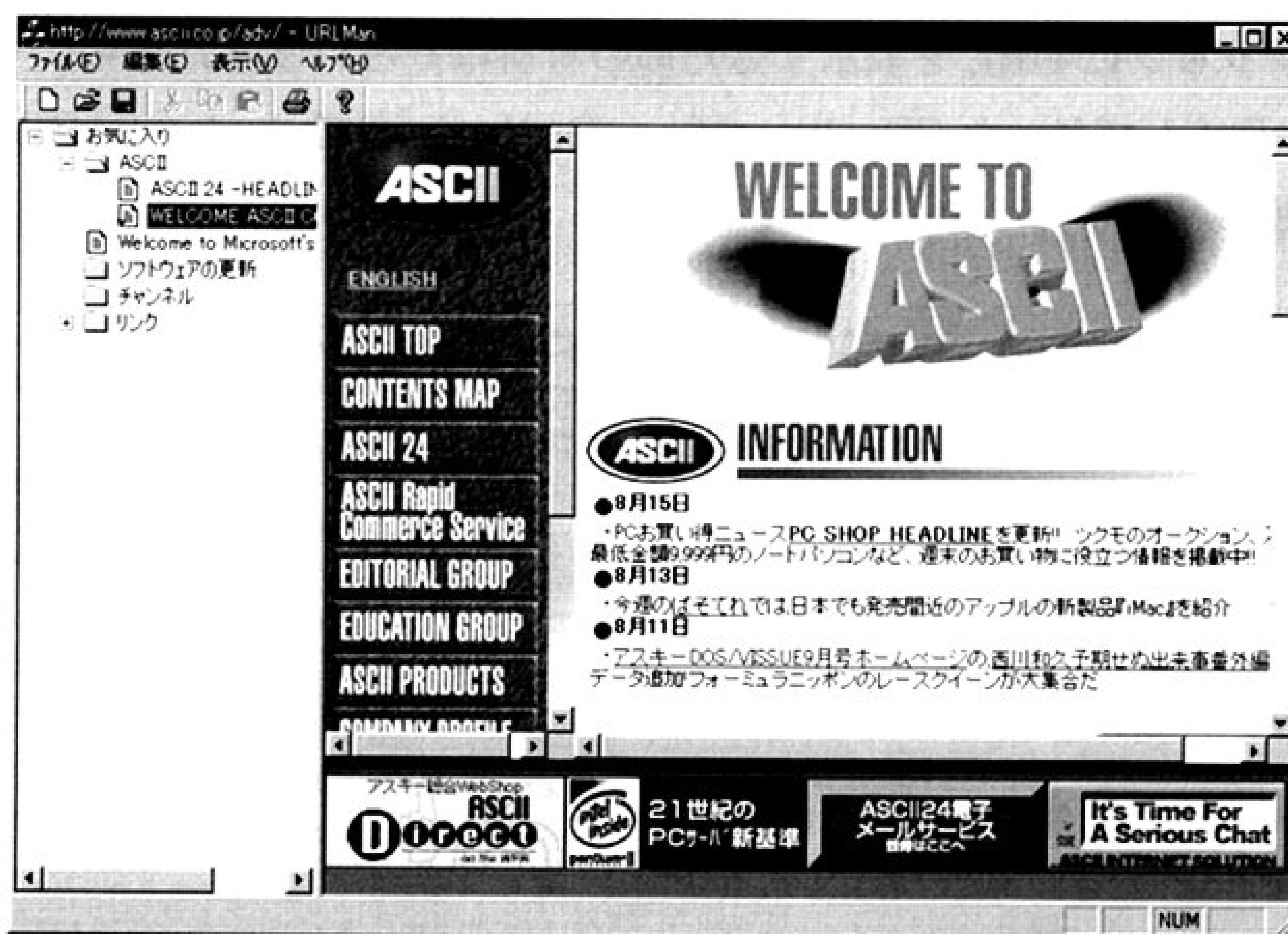


図 4-1 キャプションが設定された URLMan

4.2 ナビゲーションコマンドを追加する

さあ、いよいよ URLMan プロジェクトへの最後の作業になります。ここでは、次の 4 つのメニューコマンドを URLMan プロジェクトに追加します。

メニュー位置	オブジェクト ID
[移動 (G)]—[次に進む (F)]	ID_GO_FORWARD
[移動 (G)]—[前に戻る (B)]	ID_GO_BACK
[移動 (G)]—[停止 (S)]	ID_GO_STOP
[移動 (G)]—[最新の情報に更新 (R)]	ID_GO_REFRESH

表 4-3 追加する [移動] メニュー

まずは**図 4-2** のように、メニューエディタを使って、[移動] を含めて 5 つのエントリを追加してください。ここでの作業はすでに第 2 部で経験済みかと思いますので、細かな解説は省きます。

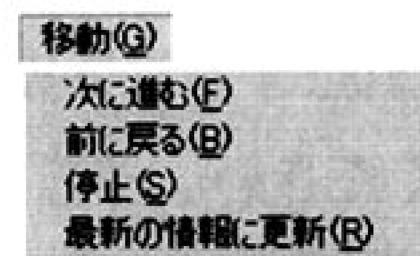


図 4-2 新規メニューコマンド

続けて、ClassWizard を使って、今作成した 4 つのオブジェクト ID に対応するメッセージハンドラを作成してください。

クラス	オブジェクト ID	メッセージ	メンバ関数
CBrowserView	ID_GO_FORWARD	COMMAND	CBrowserView::OnGoForward
CBrowserView	ID_GO_BACK	COMMAND	CBrowserView::OnGoBack
CBrowserView	ID_GO_STOP	COMMAND	CBrowserView::OnGoStop
CBrowserView	ID_GO_REFRESH	COMMAND	CBrowserView::OnGoRefresh

表 4-4 メッセージハンドラの作成

すでに予想がついているかもしれませんが、これら 4 つのメンバ関数の実装は非常に簡単です。それぞれに必要な処理はすべて CHtmlView クラスで実装されているからです。そこで、一気に実装を終えてしまいましょう。**リスト 4-2** にしたがって入力してください。

リスト4-2 メニューコマンドに対応するメッセージハンドラ(BrowserView.cpp)

```
void CBrowserView::OnGoForward()
{
  GoForward();
}

void CBrowserView::OnGoBack()
{
  GoBack();
}

void CBrowserView::OnGoStop()
{
  Stop();
}

void CBrowserView::OnGoRefresh()
{
  Refresh();
}
```

さて、これでURLマネージャは完成、といきたいところですが、まだ作業は残っています。問題はビルドして、実行してみればすぐにわかります。ツリービューからアイテムを選択して、Webページを表示してください。そして、[移動]メニューを開くとどうでしょう? メッセージハンドラを実装したはずの4つのメニューコマンドはすべてグレイ表示され、実行できない状態にあるはずです。

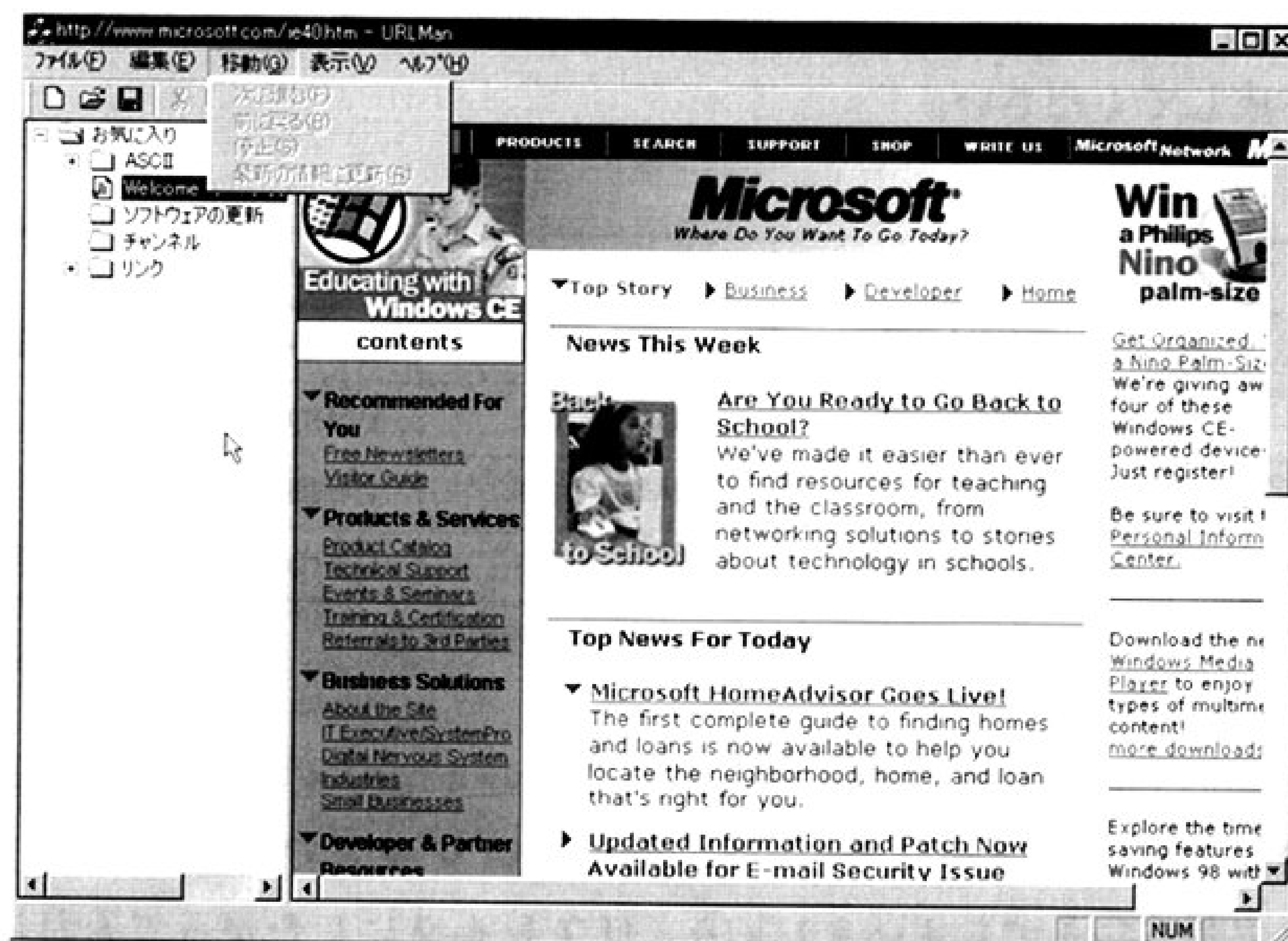


図4-3 グレイ表示されたメニュー

この現象の原因は、URLマネージャではスプリットウィンドウを使っているために、2つのビューウィンドウが使われていることにあります。第3部のMMViewを思い出してください。このアプリケーションではテキストエディタとドローツールを実装したため、2つのビューウィンドウが使われていました。そして、メニューは現在アクティブになっているビューによって、自動的に切り替わっていました。これと同じことがURLManでも起こっているのです。2つのビューウィンドウはオーバーラップしないためにわかりづらいのですが、ツリービューのアイテムをクリックした瞬間、ツリービューがアクティブになります。したがって、メニューはCURLTreeViewクラスのものに切り替わります。ここに問題があるのです。先の4つのメニューコマンドはCBrowserViewクラスにメッセージハンドラとして実装したので、ツリービューがアクティブになっているときには、［移動］メニューはすべてグレイに表示されてしまいます。こうしたしくみになっているため、一旦Webページが表示されている、CBrowserViewクラスが管理しているペインをクリックすれば、［移動］メニュー以下は使用可能になりますが、これでは実用的とはいえません。

これは問題ですが、解決は難しくありません。要は常にCBrowserViewクラスのペインがアクティブなビューになっていればよいのです。具体的には、CBrowesrViewクラスのペインが非アクティブ状態になった瞬間を検出し、再びアクティブビューとして設定すればよいのです。

ビューが非アクティブになった瞬間を捕らえるには、CView::OnActivateView仮想メンバ関数が利用できます。アクティブビューが切り替わると、すべてのビューでこの仮想メンバ関数の呼び出しが発生します。ただ、ここではCBrowserViewクラスが非アクティブビューになった瞬間を捉えればよいだけなので、CBrowserViewクラスだけで、CView::OnActivateView仮想メンバ関数をオーバーライドします。

CView::OnActivateView仮想メンバ関数は次のように定義されています。

void CView::OnActivateView(BOOL *bActivate*, **CView∗** *pActivateView*,
　　CView∗ *pDeactiveView*)

引数	**BOOL** *bActivate*	このビューがアクティブになったときTRUE、非アクティブになったときFALSEが渡される
	CView∗ *pActivateView*	アクティブになったビューへのポインタが渡される
	CView∗ *pDeactiveView*	非アクティブになったビューへのポインタが渡される

このようにCView::OnActivateView仮想メンバ関数には多くの情報が渡されますが、ここではとにかくアクティブビューが切り替わったら、CBrowserViewクラスのペインをアクティブにすればよいので、ごく簡単な処理で済みます。

ユーザーが特定のビューをクリックすればアクティブビューは切り替わりますが、プログラム中から指定したビューをアクティブにするには、CFrameWnd::SetActiveViewメンバ関数を使います。

void CFrameWnd::SetActiveView(CView* *pViewNew*, BOOL *bNotify* = TRUE)

引数	**CView*** *pViewNew*	アクティブにするビューを指定する
	BOOL *bNotify*	この関数の呼び出し後、CView::OnActivateView仮想メンバ関数の呼び出しを発生させるか指定する

それでは、CFrameWnd::SetActiveViewメンバ関数を使って、**表4-5**および**リスト4-3**のようにCBrowserView::OnActivateView仮想メンバ関数を実装してください。

クラス	オブジェクトID	メッセージ
CBrowserView	なし(CBrowserViewを選択する)	OnActivateView

表4-5 OnActivateView仮想関数のオーバーライド

リスト4-3 CBrowserView::OnActivateView仮想メンバ関数(BrowserView.cpp)

```
void CBrowserView::OnActivateView(BOOL bActivate, CView* pActivateView,
        CView* pDeactiveView)
{
  GetParentFrame()->SetActiveView(this, FALSE);
  CHtmlView::OnActivateView(bActivate, pActivateView, pDeactiveView);
}
```

CFrameWnd::SetActiveViewメンバ関数の呼び出しで、1つ目の引数がthisになっていますが、この関数呼び出しはCBrowserViewクラスのメンバ関数で行われていますから、thisはCBrowserViewオブジェクトを指しています。すなわち、メニューコマンドのメッセージハンドラが存在するCBrowserViewのペインをアクティブにすることを指示しています。次の引数がFALSEになっていますが、これはCFrameWnd::SetActiveViewメンバ関数の呼び出し後、再びCView::OnActivateView仮想メンバ関数の呼び出しが発生しないように指示しています。この実装では無条件にCFrameWnd::SetActiveViewメンバ関数を呼び出しているため、この引数にTRUEを指定すると、無限ループが発生してしまうため、これを防ぐためにFALSEを指定します。

以上でツリービューをクリックしても、CBrowserViewのペインがアクティブ状態に維持されるようになりましたが、URLマネージャを実行した直後には、ツリービューがアクティブになってしまっています。これはCMainFrame::OnCreateClient仮想メンバ関数を

実装したとき、スプリットウィンドウへビューを挿入する順番が CURLTreeView クラスが先だったからです。この順番はそのままペインの並び順につながるため、初期状態では必ず左側のペインがアクティブになっていまいます。そこで、初期状態を変更するために、CBrowserView::OnInitialUpdate 仮想メンバ関数に、次の 1 行を追加してください。これで、初期状態でも CBrowserView のペインがアクティブになります。

リスト 4-4　OnInitialUpdate 関数（BrowserView.cpp）

```
void CBrowserView::OnInitialUpdate()
{
  Navigate2(_T("about:blank"),NULL,NULL);
  GetParentFrame()->SetActiveView(this, FALSE);
}
```

さあ、今度こそ本当に URL マネージャの実装は終了です。早速ビルドして、実行してみてください。今度はちゃんと［移動］メニューが常に使用可能になったはずです。これで WWW ブラウザとしての体裁も整いました。HTML コントロールを使用することで、Web ブラウジングアプリケーションがどれだけ簡単に作成できるか、おわかりになったでしょうか？



図 4-4　URL マネージャ、ホントに完成の図

Appendix

A. 避けては通れないC＋＋言語の基礎知識

B. C＋＋のツボ：ポインタ変数

C. 非ドキュメントービュー･アーキテクチャアプリケーションの作成

A 避けては通れない C++言語の基礎知識

C++言語は非常に複雑な処理系ですから、ここでそのすべてを解説することは到底できません。また、本格的にC++言語を使いこなすには、オブジェクト指向プログラミングに関する知識も必須といえます。ただし、Visual C++という枠組みの中では、ごく限られた知識であっても、プログラムを作成することは可能です。そこで、ここではVisual C++を使ってプログラムを作成していく上で必要になる知識についてのみ説明することにします。よって、いくつかの仕様（言語上重要であるかもしれないが本書では必要としない仕様）については説明を省きます。

A.1 構造体とクラス

C++言語はC言語にいくつもの拡張を施したプログラミング言語ですが、その拡張の中心的な存在が**クラス**です。拡張された仕様のほとんどがクラスの導入に伴うものといっても過言ではありません。クラスを理解することが、C++言語を理解することの第一歩となることはいうまでもありません。

さて、それではクラスとはいったいどういうものなのでしょう。上っ面だけを見れば、クラスは構造体を拡張したものといった体裁を持っています。そこで、まず構造体とクラスを比較してみることにしましょう。

最初に、同じ処理を構造体とクラスの両方を使って書いてみます。処理内容は、指定した文字列の先頭から指定した文字数だけ表示するというものです。構造体とクラスの定義と利用例を**リスト A-1** に示します。

リストA-1 構造体とクラスの比較

```
/* 構造体の場合 */
struct sString {
  char* m_str;
  int m_len;
};

struct sString message;
message.m_str = "Hello, C++ World";
message.m_len = strlen(message.m_str);
printf("%.*s¥n", min(message.m_len, 10), message.m_str);

// クラスの場合
class cString{
private:
  char*   m_str;
  int m_len;
  void  SetLength();
public:
  void SetMessage(char*);
  void PrintMessage(int);
};

cString message;
message.SetMessage("Hello, C++ World");
message.PrintMessage(10);
```

確かに何となく似てはいますが、クラスの定義には、いろいろと見たことのないキーワードや表記がいくつかあります。見た目でわかる違いは

- クラスには、メンバに関数のプロトタイプ宣言が追加されている
- クラスには、private、public といったキーワードが使われている
- クラスのオブジェクトの定義には、class キーワードは使われていない

といったところです。もちろん、ここにあげた違いは文法上の違いでしかなく、その本質は一歩誤れば泥沼に沈みこんでしまうほど深いものです。

ところで、ここでは構造体とクラスを比較しながらクラスの説明を行っていきますが、ここでいう構造体とはあくまでもC言語での構造体であることに注意してください。というのは、C++言語では struct キーワードによって定義する構造体についても、class キーワードで定義するクラスとほとんど同じ働きをするように拡張されているからです。

A.2 メンバ関数とアクセス制御

構造体とクラスの大きな違いの1つに、クラスではメンバとして関数を定義できるということがあげられます。この関数のことを**メンバ関数**と呼びます。それに対して、従来の変数のメンバを**メンバ変数**と呼びます。構造体のメンバにアクセスするときにはアクセス演算子を使って、「＜構造体オブジェクト＞.＜メンバ＞」または「＜構造体オブジェクトへのポインタ＞->＜メンバ＞」としましたが、クラスの場合でも方法は同じです。つまり、「＜クラスオブジェクト＞.＜メンバ＞」または「＜クラスオブジェクトへのポインタ＞->＜メンバ＞」としてアクセスします。このことは、メンバ関数を呼び出すためには、そのメンバ関数が属しているクラスのオブジェクトが必要だということを意味しています。

たとえば、表示する文字列を設定する関数、SetMessage関数を考えてみましょう。構造体を使った場合には、

```
struct sString message;              /* 構造体オブジェクトの定義 */
SetMessage(&message, "hogehoge"); /* SetMessage 関数の呼び出し */
```

と、関数にオブジェクトのアドレスと設定する文字列を渡すのに対して、クラスの場合は、

```
cString message;                 // クラスオブジェクトの定義
message.SetMessage("hogehoge");  // メンバ関数、SetMessage 関数の呼び出し
```

と、オブジェクトに対してメンバ関数として定義されているSetMessage関数を実行して、文字列を設定するようになります。構造体のオブジェクトを定義するときにはstructキーワードが必要でしたが、クラスのオブジェクトを定義するときにはclassキーワードは必要ないことにも注意してください。

構造体のようにメンバの値を直接設定する代わりに、メンバ関数を使ってメンバ変数を操作すれば、各データの整合性を保つことができます。たとえば、cStringクラスでは、m_lenには文字列(m_str)の長さを保存していますが、その関係はあくまでもそのように使われることが期待されているだけです。文字列をSetMessage関数を使って設定すれば、その文字数がm_lenに保存され、データの整合性が保たれるでしょう。しかし、2つのメンバ変数の値を直接変更してしまえばその整合性が保たれる保証はどこにもありません。利用する側が、そうした使い方(ルール)を守らなければならないのです。このルールを守って、データの整合性を保ってくれるのが**メンバ関数**なのです。

ところが、メンバ関数を使わずにメンバ変数の値を直接、設定できるようでは整合性を完全に保つことはできません。そこで、C++言語にはメンバ変数を保護するために、クラス外からメンバへのアクセスを制限する方法が用意されています。各メンバに対してどこからのアクセスを許可するかを指定するために使うのが、**private**キーワードや**public**

キーワードといった**アクセス指定子**です。public キーワードを使ってパブリック属性を指定したメンバ（パブリックメンバ）には、オブジェクトさえあればプログラムのどこからでも自由にアクセスできます。一方、private キーワードを使ってプライベート属性を指定したメンバ（プライベートメンバ）は、同じクラスのメンバ関数からしかアクセスできません。つまり、他のクラスのメンバ関数やグローバル関数からはアクセスできないようになります。これらの属性は、メンバ単位に指定できます。

つまり、パブリックなメンバ関数はクラスの外からメンバ変数を操作するためのインターフェイスに相当し、プライベートなメンバ関数はクラス内部で閉じた操作を行うために利用するものと考えることができます（**図 A-1**）。

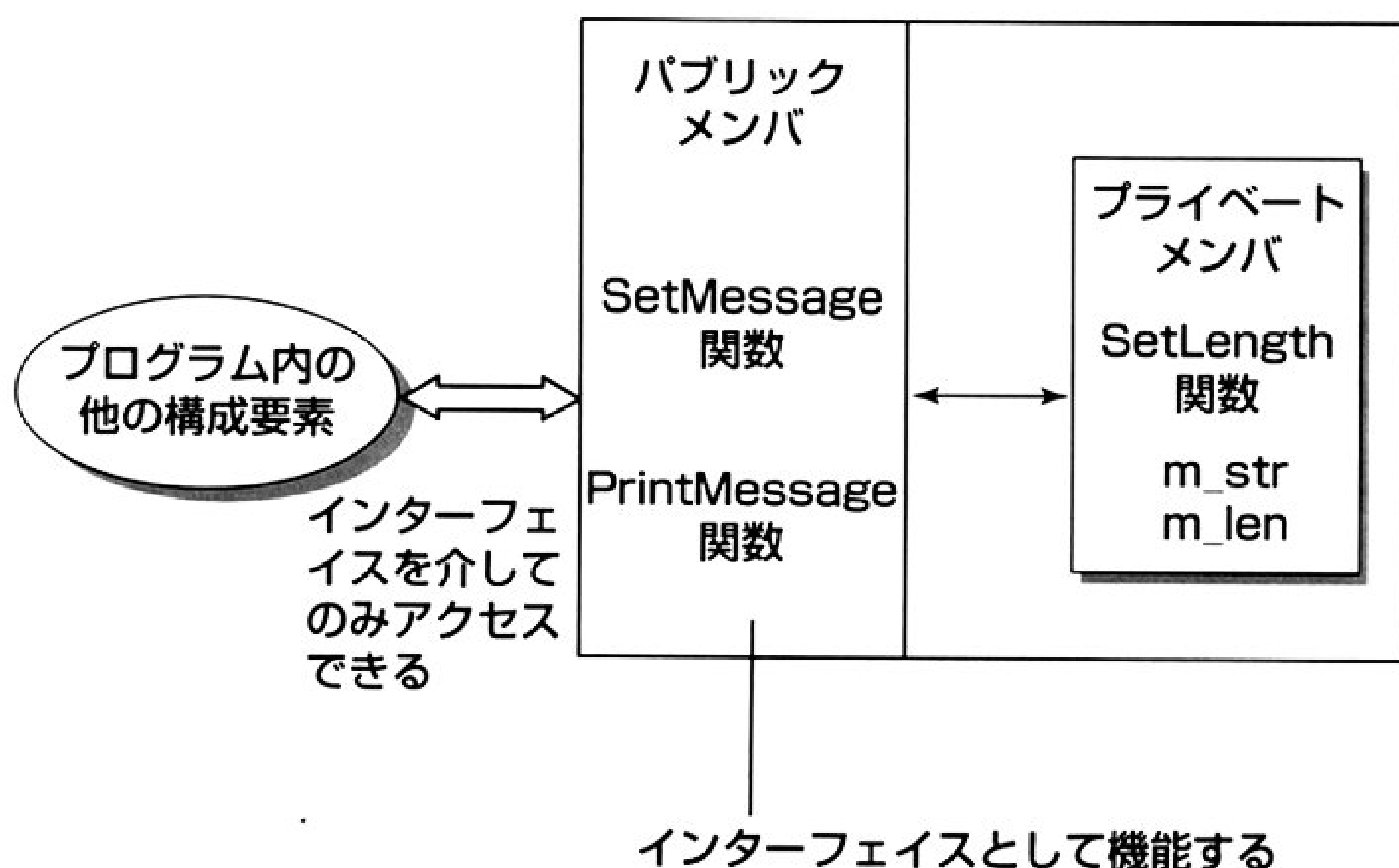


図 A-1　private と public

ところで、プライベートメンバしかないクラスがあったとしたら、そのクラスのオブジェクトを作っても、そのあとは何もできません。というのは、プライベートメンバにアクセスするためには、パブリックなメンバ関数が必要だからです。逆に、あるクラスのメンバがすべてパブリックメンバだったとしたら、誰でもメンバ変数をいじくれてしまいますから、メンバ関数なんていらなくなってしまいます。この状況は構造体とそっくりです。実際、C++言語では、構造体のメンバはアクセス指定子を使ってその属性を指定しなかった場合には、パブリックメンバとして扱われます（逆にいえば、C++言語においては構造体のメンバでもプライベート属性を持つことができることを意味する）。クラスのメンバに関しては、属性を指定しなければプライベートメンバとして扱われます。

リスト A-1 に示した cString クラスの例であれば、m_str、m_len の 2 つのメンバ変数と SetLength 関数がプライベートメンバ、SetMessage 関数と PrintMessage 関数の 2 つのメンバ関数がパブリックメンバとして定義されています。cString クラスのメンバ変

数、m_str と m_len を書き換えることができるのは、SetLength 関数、SetMessage 関数、PrintMessage 関数の 3 つのメンバ関数だけです。それ以外の関数からは参照／変更ともにできません。また、SetLength 関数を呼び出すことができるのは、cString クラスの 3 つのメンバ関数に限られます（SetLength 関数自身を含む）。パブリックメンバの SetMessage 関数と PrintMessage 関数は、オブジェクトさえあれば、プログラム中のあらゆる場所から呼び出すことできます。

A.3 メンバ関数の実装

今までの話をまとめると、メンバ関数の特徴として以下のようなことがあげられます。

- プライベート／パブリックを問わず、同じクラスのすべてのメンバにアクセス可能
- クラスオブジェクトがなければ呼び出し不可能

ここで、これらの特徴に加えてもう 1 つのメンバ関数の特徴を説明します。

- メンバ関数の中から同じオブジェクトに属しているメンバにアクセスするときには、オブジェクトを指定する必要はない

cString クラスのオブジェクト、message を使って SetMessage 関数から SetLength 関数を呼び出すことを考えてみましょう。message オブジェクトを使って、呼び出された SetMessage 関数の中から、さらに SetLength 関数を呼び出すには、やはり message オブジェクトを使って関数を呼び出す必要があります。というのは、メンバ変数の値はそれぞれのオブジェクトによって異なるからです。また m_str や m_len にアクセスする場合にも、message オブジェクトを介してアクセスしなければいけません。

しかし、SetMessage メンバ関数を記述する時点では、オブジェクトの名前はわかりっこありません。そこで、メンバ関数の中から同じクラスのメンバをアクセスする場合には、メンバ関数を呼び出したオブジェクトを使ってそのメンバ関数を呼び出すように、コンパイラが自動的に処理してくれます。このため、メンバ関数の中では、オブジェクトを指定せずにメンバをアクセスできるのです。

この処理のからくりは、**this ポインタ**という特殊なポインタを導入することによって実現されています。this ポインタは、メンバ関数を呼び出したオブジェクトを指すポインタのことで、呼び出されたメンバ関数内でのみ参照が可能です。ですから、this ポインタの型は、cString クラスのメンバ関数の中では cString*型ですし、MFC が提供する CFile クラスのメンバ関数の中では CFile*型となります。つまり、すべてのメンバ関数には、暗黙のうちに引数として this ポインタが渡されていると考えることができます。

以上のことを踏まえた上で、cString クラスのメンバ関数の実装例を見てみましょう（**リスト A-2**）。

リスト A-2 メンバ関数の実装

```
void cString::SetLength(char* s)
{
  // 文字列の長さを m_len に設定
  m_len = (s != NULL) ? strlen(s) : -1;
}

void cString::SetMessage(char* s)
{
  if (s != NULL) {
    SetLength(s);              // cString::SetLength を呼び出す
    m_str = new char[m_len];  // 文字列を保存するメモリを確保
    strcpy(m_str, s);          // 文字列をコピー
  }
}

void cString::PrintMessage(int n)
{
  if (m_str != NULL)
    // 指定した文字数と m_len の小さい方の文字数分だけ表示
    printf("%.*s¥n", min(n, m_len), m_str);
}
```

メンバ関数は、クラス定義の内部で実装（関数のコードを記述）することもできますし、クラス定義の外部で実装することもできます。外部で実装する場合には、関数名は「::演算子（スコープ解決子）」を使って、＜クラス名＞::＜メンバ関数名＞の形式で指定します。メンバ関数の名前は、クラス内部でローカルであるため、クラス定義の外でメンバ関数の実装をする際に、その関数を一意に表すためには、クラス名も含めて関数名を指定する必要があるからです。たとえば、cString クラスのメンバ関数、SetMessage 関数は「cString::SetMessage 関数」と表現されます。

A.4 コンストラクタとnew演算子

cString::PrintMessage 関数は、呼び出される前に cString::SetMessage 関数によってあらかじめ文字列が設定されていることを前提に記述されています。しかし、オブジェクトを定義した直後に、あえて文字列を設定せずに、cString::PrintMessage 関数を実行したらどうなるでしょう? オブジェクトの定義直後は、メンバ変数はまだ操作されていないため、その値は不定です。したがって、まったくわけのわからない文字列が表示されてしまうかもしれません。これは非常に危険なことですから、オブジェクトの作成直後に、メンバ変数の初期化を実行することが重要になります。**コンストラクタ**は、このような目的で利用される特殊なメンバ関数です。

C++言語では、クラスのオブジェクトを作成すると、すぐにそのクラスのコンストラクタが呼び出されます。コンストラクタはメンバ関数ではありますが、以下の2つの点が他のメンバ関数とは異なります。

- 返り値は持たない。void とも違い、型の指定すら行わない
- コンストラクタの名前はクラス名と同じ

コンストラクタであることを示すキーワードはとくにありません。そのメンバ関数がコンストラクタであることを、コンパイラは関数名と返り値がないことから判断します。

それでは、cString クラスにコンストラクタを追加してみましょう(**リスト A-3**)。このクラスのコンストラクタは、メンバ変数に初期値を代入するだけの非常に単純なものです。しかしこれだけで、オブジェクトを作成したときにはすでに、有効な値を持っていることが保証されます。

リスト A-3 コンストラクタ

```
class cString {
   ...略
public:
   cString();  //  コンストラクタ。返り値は持たない
   ...略
}

cString::cString()
{
  m_str = NULL;
  m_len = -1;
}
```

```
cString message;
message.PrintMessage(5); // 何も表示されないだけ
```

次にクラスオブジェクトの動的な作成について考えてみましょう。たとえば、以下のようにして、cStringクラスのオブジェクトをヒープ上に確保します。

```
cString* pm = (cString*)malloc(sizeof(cString));
pm->PrintMessage(5);     // 結果不定
```

オブジェクトを作成するとコンストラクタが呼び出されるといっても、このようなときにはコンストラクタが呼び出されることはありません。なぜなら、malloc関数を使って作ったオブジェクトは、あくまでもsizeof(cString)バイトのメモリ領域であって、cStringクラスのオブジェクトではないからです。単に確保したメモリ領域へのポインタをcStringクラスのオブジェクトへのポインタにキャストしているにすぎません。コンストラクタによる初期化も行われませんし、この方法はオブジェクトの動的な生成方法として適切なものとはいえないでしょう。そこで、C++言語ではオブジェクトを動的に作成するために、malloc関数に代わって、new演算子が用意されています。

先の例をnew演算子を使って書き換えると、

```
cString* pm = new cString;
pm->PrintMessage(5);     // 何も表示されないだけ
```

となります。このようにnew演算子を使ってオブジェクトを作成すると、コンストラクタが呼び出され、きちんんと初期化が行われるようになります。C++言語でオブジェクトを確保する場合には、かならずnew演算子を使うようにしてください。

new演算子は、オブジェクトの型を与えると必要なサイズのメモリを確保し、確保したメモリ領域へのポインタを返しますが、オブジェクトの型を指定する方法には、いくつかの形式があります。

1. new ＜オブジェクトの型＞
2. new ＜オブジェクトの型＞(コンストラクタの引数)
3. new ＜オブジェクトの型＞[配列のサイズ]

1.の形式では、int、char、クラス名といったオブジェクトの型を与えます。この場合は、引数を持たないコンストラクタが呼び出されます。2.の形式では、引数を持つコンストラクタが呼び出されます。3.の形式は、オブジェクトの配列を確保するときに使います。この場合には引数を持つコンストラクタを呼び出すことはできません。このときには引数を持たないコンストラクタがかならず呼び出されます。

A.5 デストラクタとdelete演算子

オブジェクトを作成した直後にコンストラクタが呼び出されたように、オブジェクトが破壊される直前には、**デストラクタ**が呼び出されます。デストラクタもやはりコンストラクタと同じように特殊なメンバ関数であり、次のような特徴を持っています。

- 返り値は持たない。void とも違い、型の指定すら行わない
- 引数を取ることはできない
- クラス名の前に~（チルダ）を付けた名前がデストラクタの名前になる

メンバ変数に動的に確保したメモリやオブジェクトがあるときには、そのクラスオブジェクトが破壊される前に動的に確保したメモリやオブジェクトを解放しなければいけません。たとえば、cString クラスでは、cString::SetMessage 関数で文字列をコピーするために new 演算子を使ってメモリを確保していますが、行儀の悪いことにこのメモリの解放をさぼっています。こういったメモリの解放などを行うのに、デストラクタを使います。

C 言語では、malloc 関数で確保したメモリ領域は、使い終わったら free 関数を使って解放しなければいけませんでした。C＋＋言語では動的にメモリを確保するには、malloc 関数ではなく new 演算子を使いますが、使い終わったメモリを解放しなければならないということはC 言語とまったく同じです。ただし、new 演算子で確保したメモリは、free 関数ではなく **delete 演算子**を使って解放します。

delete 演算子は、オブジェクトへのポインタを受け取り、そのポインタが指すオブジェクトを解放します。そして、与えられたオブジェクトの型に応じて、適切なクラスのデストラクタを呼び出します。なお、delete 演算子は演算子ですが、返り値はありません。

delete 演算子にオブジェクトへのポインタを与えるときには、ポインタが配列を指しているかどうかで以下の２つの形式を使い分けます。なぜなら、もしポインタが配列を指しているのならば、配列の要素１つ１つに対してデストラクタを実行しなければならないからです。もしオブジェクトの配列に対して最初の形式のデストラクタを実行すると、配列の先頭の要素だけしか解放されませんから注意してください。

1. delete ＜オブジェクトへのポインタ＞
2. delete ［］＜オブジェクトの配列へのポインタ＞

それでは、cString クラスにデストラクタを実装してみましょう。加えてここでは、cString::SetMessage 関数を使って文字列を設定するときに、すでに文字列が設定されている場合には古い文字列を解放するようにします。

リストA-4 デストラクタ

```
class cString {
  <省略>
public:
  CString(); // コンストラクタ
  ~CString(); // デストラクタ
};

void cString::SetMessage(char* s)
{
  if (m_str != NULL)
      delete m_str;              // 古い文字列を解放する
  if (s != NULL) {
    SetLength(s);                // cString::SetLength を呼び出す
    m_str = new char[m_len];     // 文字列を保存するメモリを確保
    strcpy(m_str, s);            // 文字列をコピー
  } else {
    m_str = NULL;
    m_len = -1;                  // m_len = -1 は文字列がセットされていないことを表す
  }
}

void cString::~cString()
{
  delete [] m_str;               // char 型の配列を確保していたため、配列として解放
}
```

A.6 クラスの継承

本節と次節では、第1部2章で作成したHelloプロジェクトを例にして説明を進めていきます。

AppWizardが生成したHello.hには**リストA-5**に示すように、CHelloAppクラスが定義されています。すでに説明したクラスの定義とは少し違い、クラス名のあとに「: public CWinApp」と付いています。これは、CWinAppクラスをもとにして、新しくCHelloAppクラスを定義する、ということを表しています。このようにあるクラスをもとにして新しくクラスを定義することを**クラスの継承**と呼びます。このとき、もとになったクラスを**基底クラス**と呼び、基底クラスを継承して新しく定義したクラスを**派生クラス**と呼びます。ここでは、CWinAppクラスが基底クラスで、CHelloAppクラスが派生クラスです。

リストA-5 CHelloAppクラスの定義

```
class CHelloApp : public CWinApp
{
public:
  CHelloApp();

// オーバーライド
  // ClassWizardは仮想関数のオーバーライドを生成します。
  //{{AFX_VIRTUAL(CHelloApp)
  public:
  virtual BOOL InitInstance();
  //}}AFX_VIRTUAL

// インプリメンテーション

  //{{AFX_MSG(CHelloApp)
  afx_msg void OnAppAbout();
    // メモ - ClassWizard はこの位置にメンバ関数を追加または削除します。
    //        この位置に生成されるコードを編集しないでください。
  //}}AFX_MSG
  DECLARE_MESSAGE_MAP()
};
```

派生クラスの特徴は、派生クラスのメンバには基底クラスのメンバがすべて含まれているということです。たとえば、CHelloAppクラスでは2つのメンバ関数(コンストラクタとInitInstance関数)しか宣言されていませんが、このクラスはCWinAppクラスから派生しているので、CWinAppクラスのメンバ変数とメンバ関数のすべてを、そのメンバに含んでいます。ただし、派生クラスのメンバ関数がアクセスできる基底クラスのメンバは、パブリックメンバだけで、基底クラスのプライベートメンバにアクセスすることはできません。

このようにクラスを継承することによって、最小限の手間で必要な機能を持ったクラスを作ることができます。

A.7 仮想関数

派生クラスのオブジェクトは基底クラスのパブリックなメンバ関数をすべて利用できますが、同名の関数が派生クラスと基底クラスの両方で定義されていたらどうなるのでしょうか？ たとえば、InitInstance関数は、CWinAppクラス（基底クラス）とCHelloAppクラス（派生クラス）の両方で定義されている関数です。このような場合は、大方の予想どおり、派生クラスのメンバ関数が優先的に呼び出されます。CHelloAppクラスのオブジェクトを使ってInitInstance関数を呼び出すと、CWinAppクラスのInitInstance関数ではなく、CHelloAppクラスで新たに定義し直したInitInstance関数が呼び出されるのです。

ところで、C++言語では以下のようなことが許されています。

● 派生クラスのオブジェクトは、キャストなしに基底クラスのオブジェクトに代入できる

ここで問題です。CHelloAppクラスのオブジェクトをCWinAppクラスのオブジェクトに代入した場合、このオブジェクトを使ってInitInstance関数を呼び出すと、どちらのクラスのInitInstance関数が呼び出されるのでしょう？

もしCWinAppクラスのInitInstance関数の宣言に**virtualキーワード**が指定されていなければ、CWinApp::InitInstance関数が呼び出されます。つまり、基底クラスのオブジェクトに派生クラスのオブジェクトを代入すると、オブジェクトを定義したときのクラスに関係なく、基底クラスのオブジェクトとして動作するということです。

ただし、InitInstance関数がvirtualキーワードを使って宣言されていたら、オブジェクトの型はCWinAppクラスでも、CHelloApp::InitInstance関数が呼び出されます。

そして、実際のところは、CWinApp::InitInstance関数はvirtual宣言された関数です。よって、この場合は、CWinAppクラスのオブジェクトにCHelloAppクラスのオブジェクトを代入しても、オブジェクトの型（CWinAppクラス）に関係なく、代入元のオブジェクトを定義したときのクラスのメンバ関数（CHelloApp::InitInstance関数）が呼び出されます。CWinApp::InitInstance関数のように、virtual宣言されたメンバ関数のことを**仮想関数**と呼びます。そして、基底クラスの仮想関数を派生クラスで再定義することを、「仮想関数のオーバーライド」といいます。

メンバ関数が仮想関数であるかどうかは、基底クラスでの関数のプロトタイプ宣言に依存します。基底クラスで仮想関数として宣言されていれば、派生クラスでも仮想関数として扱われます。CHelloAppクラスの定義（**リストA-5**）ではInitInstance関数を仮想関数として宣言していますが、この場合のvirtualキーワードはあってもなくても同じということです。仮想関数を宣言するということは、「継承されることを前提として設計されたクラスにおいて、その派生クラスのメンバ関数を呼び出すための抜け道を用意しておく」

ことであるといえます。

加えてもう1つ、仮想関数には重要な仕様があります。それは、「仮想関数を使って実際に呼び出される関数は、プログラムを実行するまで決定されない」ということです。たとえば、CWinAppクラスのオブジェクトを使ってInitInstance関数を呼び出す場合、そのオブジェクトが本当にCWinAppクラスのオブジェクトなのか、派生クラスのオブジェクトなのかによって、実際に呼び出す関数が変わってくるからです。このため、すでにコンパイル済みのコードであっても、仮想関数であればあとから作成した関数を呼び出すことができるのです。このような、実行中に呼び出す関数を決定することを**動的結合**(Dynamic Binding)と呼びます。これに対して、コンパイル時に呼び出す関数を決定することを**静的結合**(Static Binding)と呼びます。

A.8 その他

本節ではクラスとは直接関係のない、C++言語で追加された仕様を3つ取り上げることにします。

● オーバーロード

C言語では、同じ名前の関数を1つのプログラム内で複数作ることはできませんでした。一方、C++言語では、メンバ関数の名前はクラス内でローカルなので、クラスが違えば、同じ名前の関数を作ることができるのはすでに述べたとおりです。それどころか、同じクラス内でさえ、同じ名前の関数を作ることも可能です(クラス内ではなくても、つまりグローバル関数でも可能)。このように、同じ名前空間の中で同じ関数名を持つ関数を定義することを**関数のオーバーロード**と呼びます。

もちろん、関数名はおろか返り値の型や引数まですべて同じ関数を複数定義することはできません。C++言語では関数名/返り値/引数と3つの条件がマッチした関数が呼び出されるからです(C言語では関数名だけを使って呼び出す関数を特定していた)。したがって、デストラクタのように引数も返り値も持たない関数は、オーバーロードできません。

● デフォルト引数

C++言語では、省略可能な引数を持った関数を定義できます。このような引数のことを**デフォルト引数**と呼び、引数の指定を省略して関数を呼び出したときには、プロトタイプの宣言時に指定した値が引数として使われます。

デフォルト引数は関数のプロトタイプ宣言のときに次のようにして設定します。

```
void Setup(char* name = NULL, int value = -1);
void Setup(char* name, int value = -1);
```

しかし、次のような宣言はできません。デフォルト引数は、引数の末尾から順に抜けがないように設定する必要があります。

```
void Setup(char*name = NULL, int value);
```

最初の例にあげたプロトタイプ宣言をした場合、次の関数呼び出しはすべてこのSetup関数を呼び出すことになります。足りない引数については、デフォルト引数が与えられたとみなされます。

```
Setup();                     // Setup(NULL, -1); と同様
Setup("Visual C++");         // Setup("Visual C++", -1); と同様
Setup("Visual C++", 8);      // Setup("Visual C++", 8);
```

しかし、次のような関数呼び出しはできません。デフォルト引数を持った関数を呼び出す場合には、引数の先頭から順に抜けがないように与える必要があります。

```
Setup(, 8);
```

なお、デフォルト引数を持った関数を定義する場合には、そのことをはっきりさせるために、次のようにコメントでデフォルト値を記述しておくとよいでしょう。

```
void Setup(char* name /* = NULL */, int value /* = -1 */)
{
  <Setup 関数で処理する内容>
}
```

● 参照

C++言語では、ポインタによく似た**参照**が新たに導入されています。詳しい説明を始める前に、ポインタと参照の両方を使って、同じ処理を行うプログラムを見てください。

リストA-6 参照とポインタ

```
// ポインタを使用
int var = 1;
int* ptr = &var;
printf("%d", *ptr); //1 を表示
*ptr=2;
printf("%d", var);  //2 を表示

// 参照を使用
int var = 1;
```

```
int& ref = var;
printf("%d", ref);  // 1を表示
ref = 2;
printf("%d", var);  // 2を表示
```

リスト A-6 を見た限りでは、参照は「ポインタが指すオブジェクトにアクセスするときに使う間接演算子（*）を不要にしたもの」または「指定した変数の別名」であるといえます。これが何の役に立つのでしょう？

参照は主に、関数の引数と関数の返り値に活躍します。参照を使って引数を渡した場合は、アドレスを渡しているにもかかわらず、その関数の中では間接演算子やアクセス演算子（->）を使わなくても、その引数を参照したり、変更できます。つまり、参照には「アドレス渡しによるオーバーヘッドの減少」と「プログラムの簡潔な記述」という2つの利点があるといえます。ただし、参照であるかどうかは関数の定義を見なければはっきりしないため、関数の中で値を変更できるのかどうかはっきりしないという欠点があります。値を変更する可能性がある場合には、できるだけコメントではっきりさせておくとよいでしょう。

リスト A-7　参照とポインタを使った引数の参照渡し

```
struct loc {
  int x, y;
}

void plusone(loc& ref)     // 参照型を引数に取る
{
  ref.x++;
  ref.y++;
}

void plusone(loc* ptr)     // ポインタを引数に取る
{
  ptr->x++;
  ptr->y++;
}

loc value;
plusone(value);    // 参照型を引数に取る関数呼び出し
plusone(&value);   // ポインタを引数に取る関数呼び出し
```

参照を使った関数の返り値に関しても同様な利点があるのですが、さらにもう1つ重要な利点として、関数の呼び出しを代入式の左辺値として利用できることがあげられます。これは、本書では扱っていませんが、[]演算子や()演算子のオーバーロードを行う場合に簡潔な表現をするために重要になります。

リストA-8 参照による関数の返り値

```
loc& dist(loc& ref);        // dist関数の宣言
loc value1, value2;         // value1, value2を定義
value2 = dist(value1);
dist(value1) = value2;      // 関数の返り値に直接値を代入
```

B　C++のツボ：ポインタ変数

本書で初めてC++言語にチャレンジしている読者の方は、ポインタの扱いに苦労してはいないでしょうか。しかし、心配はいりません。C++プログラマなら、誰もがポインタのことで頭を抱えたことがあるハズです。抱えたことがないという人は、天才か、はたまた実はC++についてホントは理解していないかのどちらか、というほどのモノなのです。そこで、Appendix Bでは、話題をこの難関に絞り、ポインタについて解説します。

B.1　変数とメモリ

C++言語で変数を使い始めるには、

```
CString s;
int i;
```

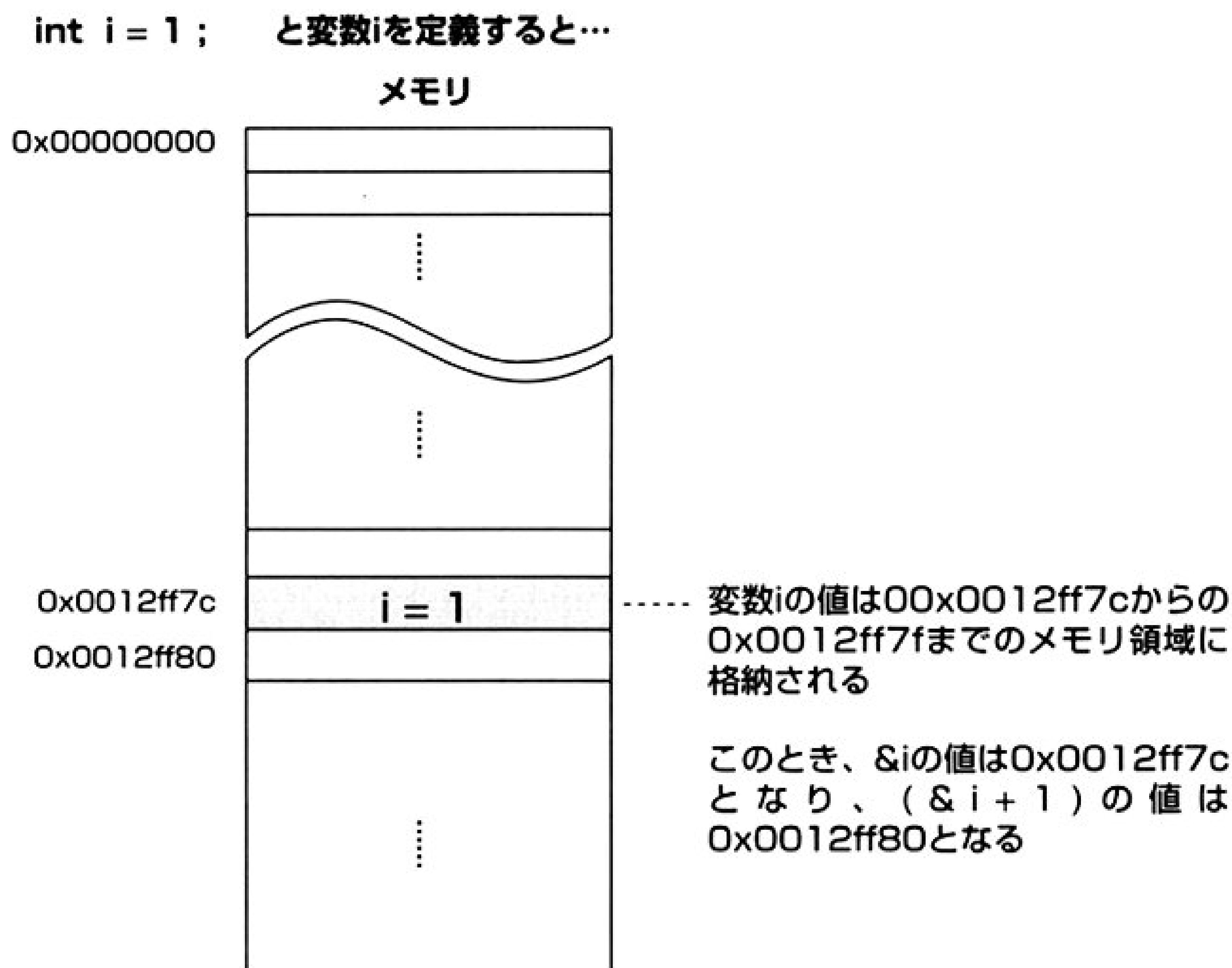
といったように変数の定義を行わなければなりません。これはいうまでもありませんが、それでは変数の定義とはいったい何を意味するのでしょう？ 数値演算やライブラリ関数の呼び出しのように、何か処理が行われるわけではありませんが（ただし、クラスオブジェクトの定義ではコンストラクタが起動される）、単に形式的に必要とされているわけでもありません。

変数とは情報を格納するための入れ物ですから、格納するためのスペースがコンピュータのメモリ上に必要です。このスペースを確保する作業が変数の定義にほかなりません。スペースを確保して初めて変数に情報を格納できるようになるため、変数を使うためには定義が必要というわけです。確保されるスペースのサイズは変数の型に依存します。たとえば、LONG型変数ならば4バイト、SHORT型変数ならば2バイトという具合です。またCString型やCWnd型などのクラスオブジェクトを定義した場合には、こうしたクラスにはたくさんのメンバ変数があり、それらすべてのメンバ変数のためにいっせいにスペースが確保されるため、数十バイト、数百バイトといった規模でメモリが確保されます。

ところで、このときメモリ上のどこにスペースが確保されるのでしょう。メモリにはアドレスと呼ばれるインデックスが1バイト単位でふられています。つまり、変数のために確保されたメモリ上のスペースは、その先頭アドレスを調べれば位置がわかります（**図B-1**）。そして、変数が割り当てられているアドレスを調べるには、&sや&iのように&演算子を使います。ただし、こうして得られたアドレス値を数値として表示しても、あまり意味はありません。変数のために必要なメモリスペースは自動的に適切な場所に確保されるため、プログラマが意識する必要はないからです。たとえば、変数iがアドレス0x0012ff7cから続く4バイト（0x0012ff7fまで）に格納されていたとしましょう。この場合、たとえば、

```
if (i == 0) ...
```

のようにして、変数iを参照すれば、アドレス0x0012ff7cに格納されているデータが参照されますが、プログラマにしてみれば、単に変数iをアクセスしたのであって、アドレス0x0012ff7cに格納されている値を参照したことなど知っている必要はありません。



図B-1 変数とメモリ

ただし、アドレス0x0012ff7cのように具体的なアドレス値を知っている必要はなくとも、アドレス値を取り出すことには意味があります。たとえば、(&i + 1)という式は、&iの値に関係なく、変数iが格納されているメモリスペースの次のアドレスを表します。もし配列のように連続したメモリスペースにデータが格納されているとしたら、変数iのアドレスを基準にして、相対的なインデックスを指定して、順番にメモリの内容をアクセスすることができるというわけです。後述するように、この他にもデータが格納されている

アドレスを扱う利点は多岐にわたります。そこで、C／C++言語にはアドレスを値に持つ変数型が用意されています。そして、この変数型の変数のことこそ、皆さんの頭を悩ませているポインタなのです。

B.2 ポインタ、基礎の基礎

なぜポインタの扱いは難しいとされるのでしょうか。理由の1つは、ポインタの値が正確に想定された範囲になければ、即原因究明の難しい誤動作が起きるからです。整数値の演算ならば、たとえば座標の計算に間違いがあっても、とっぴょうしもないところにウィンドウが表示されたりするかもしれませんが、即アプリケーションが異常終了することはありませんし、問題箇所の見当をつけることは難しくはないでしょう。しかし、ポインタの値が誤っている場合は、実行するたびに結果が異なったり、問題箇所とはまったく関連のなさそうなコードで異常動作が発覚したり、問題のあるポインタ操作を行ってからしばらく経たなければ異常動作が発覚しなかったりと、わずかなミスがやっかいなトラブルを引き起こします。

そうしたポインタ特有の問題に悩まされないためにも、まずはポインタを扱う作法の基礎をしっかり抑えておきましょう。ここでは、ポインタ変数の定義と、ポインタによるメモリの間接参照について解説します。

●ポインタの定義

ポインタはint型やchar型などの変数型とは異なり、すべての変数型が持つ属性と考えることができます。つまり、int型のポインタ、char型のポインタ、さらにCString型のポインタなど、あらゆる変数型／クラス／構造体について、ポインタが存在します。int型へのポインタを使ってデータを参照すると、そのポインタが指し示しているアドレスにはint型のデータが格納されているものとして処理されます。変数型によってデータのサイズ（sizeof演算子で取得可能）は異なり、また異なる変数型のデータをポインタで参照してしまうと本来とは異なる内容で読み出されてしまいますから、ポインタを定義するときには参照先の変数型を指定しなければなりません。いずれの変数型へのポインタであろうとも、ポインタに格納されるのは常にアドレスですが、ポインタを使ってデータを参照するためには変数型が必要なのです。

このため、ポインタ型変数を定義するには、「変数型」+「ポインタを表す記号（アスタリスク、*）」の形式で記述します。たとえば、int型へのポインタ変数piを定義するならば、

```
int* pi;
```

と記述します。同じように、クラスCStringへのポインタならば、

```
CString* ps;
```

のように記述します。通常の変数定義と異なり、変数型と変数名の間に*を指定することで、ポインタ変数を定義することを示しています。こうして、指定した変数型へのポインタ変数が定義されます。

このとき定義されるのはあくまでもポインタ変数であり、指定した変数型の変数が定義されるわけではないことに注意してください。すなわち、「int* pi」としても、int型へのポインタを格納するメモリスペースが確保されるだけで、int型の数値を格納するためのメモリスペースは確保されません。また、「CString* ps」としても、CString型へのポインタ変数が定義されるだけで、CStringクラスのコンストラクタは起動されませんし、この時点ではまだCStringクラスのメンバへアクセスすることもできません。

つまり、ポインタ変数を定義したあとには、必ず適切なアドレスを代入して初期化しなければ、ポインタ変数を使ってメモリ上のデータをアクセスすることはできないということです。

初期化せずにポインタを使って読み書きすると、何が起こるのでしょう。C／C＋＋言語では、グローバル変数を定義すると初期値は0に、ローカル変数を定義すると初期値は未定になります。ポインタの値が0ということは、アドレス0x0を指し示すことを意味しますが、このアドレスはシステムによって保護されているメモリ領域であるため、読み書きを行おうとすると、アクセス違反としてプログラムは異常終了してしまいます。また、値が未定のままのローカルポインタ変数を使って読み書きを行うと、グローバル変数と同じように、システムが保護しているメモリ領域をアクセスして異常終了するか、まったく予想のつかないメモリ領域をアクセスすることになります。前者は異常終了するだけまだましです。後者の場合は何が起こるか予想がつきません。読み出したときには、プログラムを実行するたびに異なる、無意味な値が取り出され、プログラム自体は何事もなかったかのように処理が続けられてしまうやもしれません。また、初期化されていないポインタが指すメモリ領域に書き込みを行えば、運悪くメモリ上のデータを破壊してしまうかもしれません。こうしたバグはもっとも発見の難しい類のバグの1つに数えられるほどやっかいなものです。したがって、初期化していないポインタを使って、メモリを参照しては絶対にいけません。

では、ポインタを初期化するにはどうすればよいのでしょうか？ ここで出てくるのが、さきほど紹介した&演算子による、変数のアドレスの取得です。たとえば、

```
int i = 1;
int* pi;
```

として、int 型へのポインタ pi と int 型変数 i があったとすると、

```
pi = &i;
```

とすることで pi に、i のアドレスが設定されます（図 B-2）。これはもっとも簡単なポインタの初期化の方法の 1 つです。このほかにも、ポインタを初期化する方法はいくつかありますが、それらについてはまた後ほど説明をします。

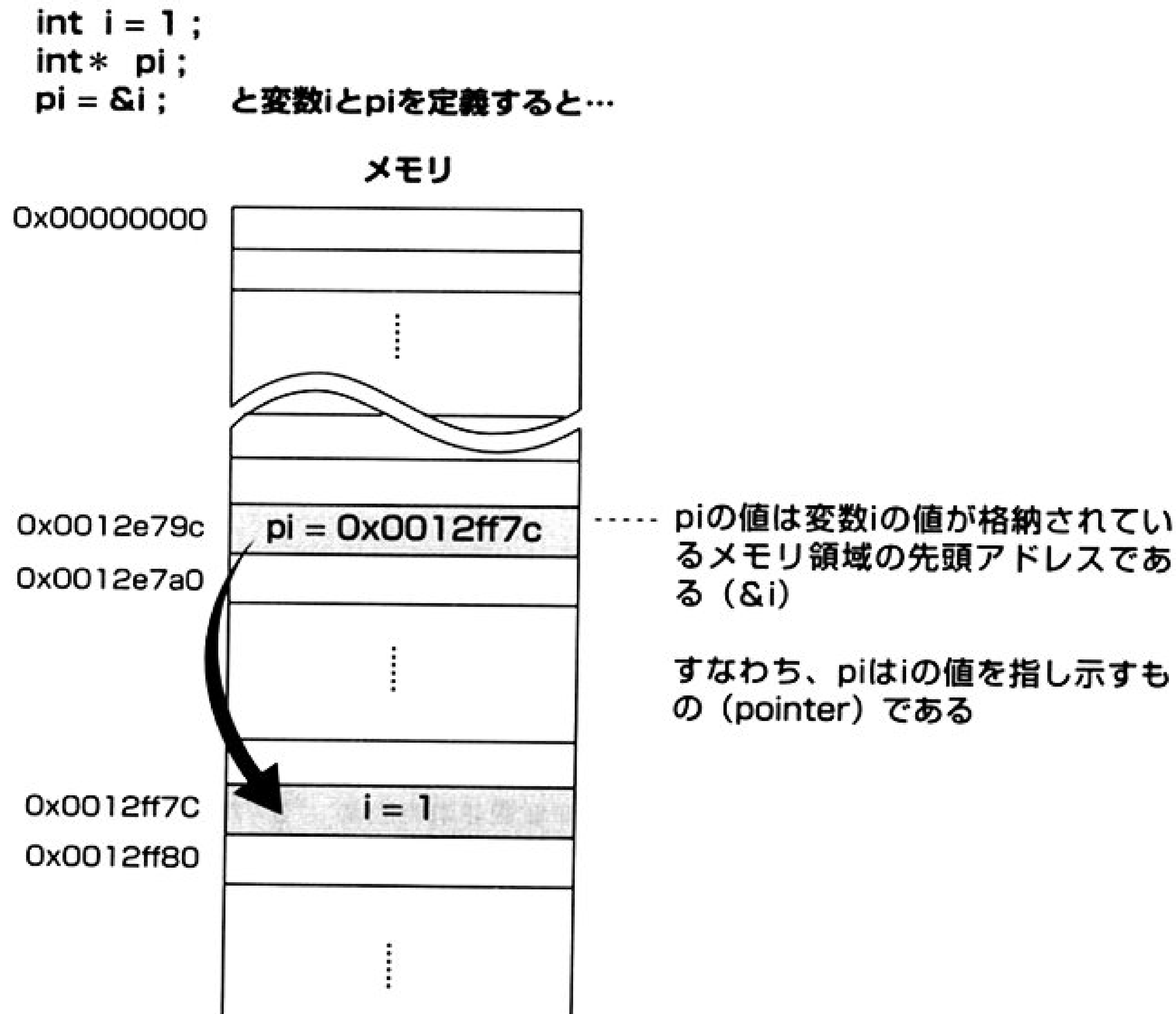


図 B-2　ポインタと変数の関係

● ポインタによるメモリの間接参照

ポインタを初期化するにはいくつかの方法があるので、ひとまずここでは置いておいて、何らかの手段ですでに初期化が行われているポインタを使ったメモリアクセスに話題を移しましょう。

次のようにポインタ p が定義され、何らかの値（アドレス）が p に代入されている場合、

```
int* p;
p = ...;
```

p に格納されているのはアドレス値ですから、

```
int i = p; //エラー!!
```

のように単にpの値を参照しただけでは、ポインタpが指し示すメモリへアクセスできるわけではありません。ポインタが指し示すメモリへアクセスするためには、ポインタ名の前にアスタリスクを指定します。ポインタ変数を定義するときにもアスタリスクを使いましたが、意味的にはまったく異なります。定義の時のアスタリスクは、ポインタ変数の定義を表すものでしたが、ここでのアスタリスクは、ポインタ変数を通して間接的にメモリの内容を参照することを表します。

たとえば、ポインタを使ってメモリにアクセスすると、次のようになります。ここでは、int型変数iが格納されているアドレスを&iによって取り出し、これを代入することでポインタpを初期化しています。したがって、以後*pを参照すれば変数iの値が参照され、*pの値を書き換えれば変数iの値も書き変わります。また、変数iの内容を書き換えれば、*pで参照される値も変わります。

```
int i = 1;
int* p = &i;          /* 変数iが格納されているアドレスをポインタpへ代入 */

printf("%d¥n", *p); /* *pによって、変数iの内容が参照されるため、"1"が表示される */

i = 2;
printf("%d¥n", *p); /* 変数iの値が変更されたため、"2"が表示される */

*p = 3;               /* *pによって変数iの内容が3に書き変わる
                         ポインタpに格納されているアドレス値に影響はない */
printf("%d¥n", i);  /* 前行によって変数iの値が変更されたため、"3"が表示される */
```

このようにポインタを使うと、メモリ上のデータを間接的に参照できるようになりますが、アスタリスクの有無によって変数型が変化することに注意してください。たとえば、前述の例ならば、次のようになります。

参照方式	変数型
p	int*型
*p	int 型

ポインタ変数はアドレス値を格納する変数型ですが、アスタリスクを付けると、参照先の変数型に変化します。

次にクラスや構造体へのポインタの使い方ですが、基本的にはint型などへのポインタと同じです。アスタリスクを前置して参照すれば、参照先のクラスオブジェクトや構造体として利用できます。

```
CString* s;
```

参照方式	変数型
s	CString*型
*s	CString 型

ポインタを介して、クラスや構造体のオブジェクトのメンバへアクセスするには、次の2通りのどちらかの方法を使います。一般的には、前者の->演算子を使った形式が使われます。

```
s->GetLength()
(*s).GetLength()
```

これまでのアスタリスクを使った形式にのっとれば、後者の形式となりますが、この場合*sを囲むように括弧が必要になります。これは、アスタリスクよりもメンバ参照ドット(.)の方が優先順位が高いため、「*s.GetLength()」と記述すると、コンパイラがこれを「*(s.GetLength())」と解釈してしまうためです。ここでは変数sはポインタですから、直接メンバ参照ドットを使ってメンバにアクセスすることはできないため、エラーになってしまいます。しかし、いちいちカッコでくくるのはわずらわしいので、一般的には前者の->演算子が使われるのです。

B.3 ポインタ型変数のメリットと初期化

ポインタを使う理由あるいはメリットとはなんでしょう。

消極的な理由としては、MFCで提供されている関数でポインタが使われているから、というものがあります。たとえば、CView::OnDrawメンバ関数の引数にはCDCクラスオブジェクトへのポインタが使われていますし、AfxMessageBox関数では表示するメッセージのためにポインタを引数に使います。

もちろん、ライブラリでポインタが使われていることには明確な理由があります。おおざっぱに表現すれば、ポインタを使うことで処理速度が向上し、コードが短くスマートになる効果が期待できるからです。また、必要なときに必要なだけメモリを確保する、動的なメモリ管理が可能になる点もポインタを利用する大きな理由の1つです。

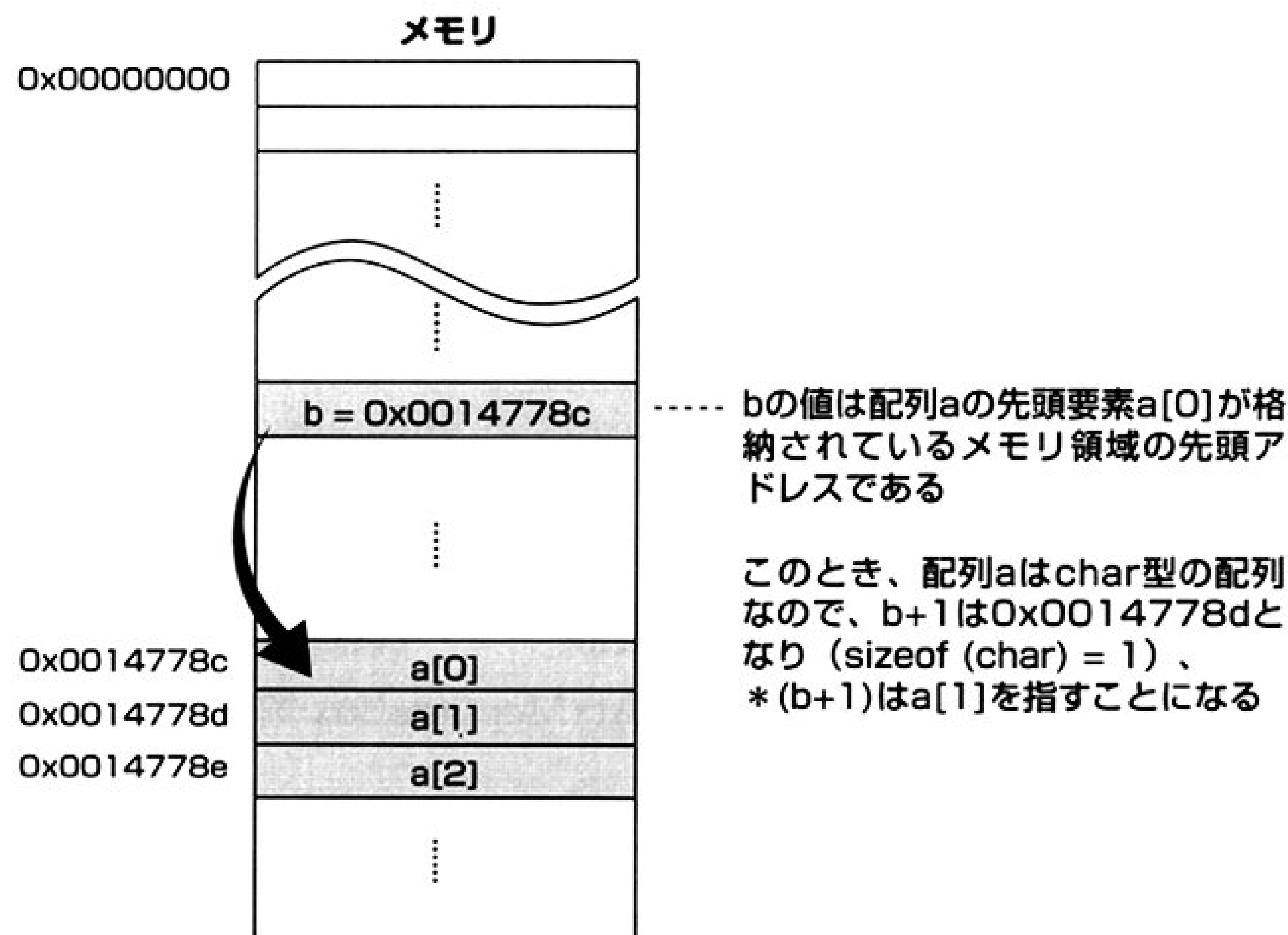
それでは、具体的にポインタを利用するメリットをケース別に解説することにしましょう。ここでケース分けのポイントになるのが、前節からの持ち越しになっているポインタの初期化方法です。ポインタを利用する前には初期化が欠かせないのは前述したとおりですが、この初期化方法にからめて、ポインタを使うメリットを解説します。

●配列

C＋＋言語の配列とポインタは兄弟のようなものです。C++言語では厳しい型チェックが行われるため、異なる型間での代入や比較は不正な処理とされたり、警告を受けたりしますが、同じ型のポインタと配列ならば、ポインタ変数へ配列をそのまま代入できます。そして、これがポインタを初期化する方法の1つとなります。

たとえば、char型の配列変数aと、char*型のポインタ変数bがあったとき、次の処理によって、ポインタbは配列aによって初期化されます。もっと具体的に表現すると、配列aを定義することによって10バイトのメモリが確保されるわけですが、この10バイトの先頭アドレスがポインタbに代入されます（図B-3）。

```
char a[10];
char* b = a;   /* 配列変数を配列要素へのアクセスに使わなければ、ポインタと
                  同じアドレス値を持つ定数として扱われる
               */
```



図B-3　配列とポインタ

したがって、次の左右の式はまったく同じ値を返すことになります。

```
a[0]   *b
a[1]   *(b + 1)
a[2]   *(b + 2)
 ⋮        ⋮
```

ただし、逆に配列変数 a に対してポインタ b の値を代入することはできません。ポインタは変数ですから値を変更できますが、配列変数は配列の要素が変数であって、配列そのもの、つまり配列のために確保されているメモリ上の位置は定数にすぎないからです。

さて、こうしてポインタを利用して配列を扱えることを述べましたが、配列をあらためてポインタでアクセスする理由はなんでしょう。

まず配列を定義すると、固定長のメモリスペースを手軽に確保できます。すでに述べたように、ポインタを定義するだけでは、データを格納するためのスペースを確保できませんから、まずは配列を定義します。

次に配列の要素にアクセスするときですが、ここでは例として、関数 lookup を定義して、この関数の中で配列にアクセスする処理を考えてみましょう。

```
char a[10]
int result;

result = lookup(a, 3);

int lookup(char array[], int index)
{
  /* 何か配列に対して処理をして、戻り値を返す */
}
```

lookup 関数の定義を見るとわかるように、配列変数に加えて、要素を指定するためのインデックスを int 型の変数で受け取っています。

次にポインタを使った同様な処理を見てみましょう。

```
char a[10];
char* p = a;
int result;

result = lookup(p + 2);

int lookup(char* p)
{
  /* 上述の lookup 関数と同じ */
}
```

どうでしょうか。今度はポインタ変数を 1 つ渡すだけで済んでいます。lookup 関数の内部では配列 a にアクセスしていることはわかりませんが、とくに問題はありません。つまり、配列の要素を指定するために、配列としてアクセスするには配列変数とインデックスの 2 つの値が必要ですが、ポインタならば、ポインタ変数 1 つで済むということです。実にささいなことですが、引数に配列の要素を 4 つも 5 つも指定する関数ならば、よりその差は際だってきますし、戻り値として配列の要素を返したいときには、1 つの変数で表現できる点は見逃せません。

● 文字列とポインタ

C++言語では、次のようにして、文字列でchar型配列を初期化できます。

```
char str[] = "Hello Visual C++ World";
```

つまり、文字列のデータ型はchar型配列であることを意味しています。そして、前節で述べたように、char型配列とchar型ポインタは同じデータ型として扱うことができるので、次のようにして、char型ポインタを文字列で初期化することもできます。

```
char* str = "Hello Visual C++ World";
```

● &演算子

すでに解説したように、&演算子を使えば、変数が格納されているアドレスを取り出すことができます。つまり、ポインタによるメモリの間接参照を行う*演算子のちょうど逆の働きをするわけです。このため、次のようにして、&演算子を使ったポインタの初期化が可能です。

```
char a;
char* b = &a; /* ポインタbを変数aが格納されたアドレスで初期化 */
```

もっとも、本当にこのとおりのコードを記述しても意味はありません。これだけでは、char型変数aを使おうが、ポインタ変数bを使おうが、結果はまったく同じだからです。この&演算子によるポインタの初期化は、主に関数呼び出し時に使います。

例として次のコードを示します。

```
CRect r(0, 0, 200, 200);
pWnd->MoveWindow(&r);
```

CWnd::MoveWindowメンバ関数は、次のように引数にCRectクラスへのポインタを受け取るように定義されています。

```
void CWnd::MoveWindow(CRect* pr)
```

したがって、上述したCWnd::MoveWindowメンバ関数の呼び出し時には、

```
pr = &r;
```

に相当する処理が行われるというわけです。こうして、ポインタ変数prの初期化が関数呼び出しと同時に行われます。

ところで、関数呼び出しになぜポインタを使うのでしょう。これには3つの理由があります。

1つ目は、関数呼び出しを高速に行うためです。C++言語では通常、関数を呼び出すと

き、引数のコピーが作成されます。たとえば、次のように定義されたProcessWindow関数があったとしましょう。この関数では、CWndオブジェクトをポインタではなく、通常のCWndオブジェクトとして受け取ります。

```
void ProcessWindow(CWnd w)
```

この関数を呼び出すと、まずCWndオブジェクトである変数wがローカル変数として新たに作成され、その後呼び出し時に引数として指定したCWndオブジェクトのメンバ変数の値がすべて変数wにコピーされることになります。したがって、関数呼び出しのたびに、CWndクラスのコンストラクタが呼び出され、数十バイトのメンバ変数がコピーされます。これに対して、次のように定義されたProcessWindowP関数ならば、わずかにsizeof(CWnd*) = 4バイト(Pentiumプロセッサの場合)のポインタ変数がコピーされるだけで関数呼び出しが実行されます。

```
void ProcessWindowP(CWnd* w)
```

膨大な回数が繰り返される関数呼び出しでのコピー処理時間は無視できるものではありません。引数がint型変数であったり、char型変数であったりした場合には、コピーされるデータサイズはポインタとほとんど変わらないため、ポインタで渡す必然性はありませんが、サイズの大きなクラスオブジェクトや構造体の場合には、ポインタで渡すと関数呼び出しが高速化されるのです。

もう1つの理由も、この引数のコピーに関連します。関数の引数へ「コピー」によって値が渡るということは、関数呼び出しに指定した変数と、受け取った側の変数は同じ値を持つ、別の変数ということです。たとえば次のように前述したProcessWindow関数を呼び出したとしましょう。このとき、変数wndとProcessWindowに引数として渡されたwは、内容こそ同一ですが、2つの独立した変数で、変数wはProcessWindow関数の処理が終了すると同時に削除されてしまいます。

```
CWnd wnd;
ProcessWindow(wnd);
```

ここで問題なのは、ポインタを使わずに関数へ値を渡したときには、呼び出しに指定した変数は何も操作できないということです。この例ならば、ProcessWindow関数内部の変数wを操作しても、変数wndには何の影響もありません。関数内部での処理結果を残したければ、戻り値として値を返すしかありませんが、それでは1つの値しか返せません。

しかし、次のように、ポインタバージョンであるProcessWindowP関数を呼び出したときには、関数内部での変数wへの操作はそのまま変数wndへの操作となります。ポインタを使うことによって、あたかも関数呼び出し時の引数コピーが行われなかったかのように、呼び出し側で定義された変数を操作できるようになるのです。

```
CWnd wnd;
ProcessWindowP(&wnd);
```

さらにもう1つ、C++に特有のスライシング問題と呼ばれる問題もあります。上に示したProcessWindow(CWnd)にCWndクラスから派生したクラスのオブジェクト(たとえばCView)を渡したとするとどうなるでしょうか？ ProcessWindow(CWnd)の引数の型はCWnd型ですから、関数の呼び出し時にCWndクラスのオブジェクトが作成され、そこにCViewクラスのオブジェクトのうち、CWndクラスに相当する部分だけがコピーされます。そして、他の部分は切り落と(スライス)されてしまうのです。

このとき、ProcessWindow関数のなかで、引数の実際の型に依存した処理(仮想メンバ関数の呼び出しなど)が行われていた場合、本来はCViewクラスに依存した処理が行われることを意図して、プログラマがProcessWindow関数にCViewクラスのオブジェクトを渡しても、渡されるのはあくまでCWndクラスに切り落とされたオブジェクトですから、CWndクラスに対応した処理が行われてしまうのです。

こうしたことを避けるためにも関数の引数をポインタで渡すことが重要になってきます。

● new演算子

上記3種類のポインタ初期化方法は、すべてすでに確保された変数のアドレスを取得して、ポインタに代入していました。つまり、すでに存在する変数の別名としてポインタを利用したわけです。対して、new演算子によるポインタの初期化は、新規にメモリを確保して、そのメモリの先頭アドレスをポインタに代入する点でこれまでとは異なります。つまり、new演算子を呼び出した直後では、その値を代入したポインタ以外には、確保したメモリ領域にアクセスする手段はないということです。

冒頭で、変数の定義とはデータを格納するためのメモリスペースを確保することだと述べました。つまり、new演算子は変数定義のもう1つの形ともいえるでしょう。ただし、明示的にdelete演算子によってメモリ領域を解放することができる、そして明示的に解放しない限りはメモリ領域は維持され続ける点が大きく違います。

変数にはグローバル変数とローカル変数があります。グローバル変数はプログラムの起動と同時にメモリ領域が確保され、プログラムが終了するまで維持されます。ローカル変数は、関数呼び出しと同時に確保され、関数の実行が終了すると自動的に削除されます。いずれにしろ、変数が存在している期間をプログラマが制御することはできません。また、確保するメモリ領域のサイズ(たとえば配列の要素数)は、固定長でなければなりません。すなわち、new演算子を使わないとしたら、プログラムが扱えるデータサイズをコンパイル時に決定しなければならないため、無意味に膨大なメモリ領域を確保して余らせるか、足りなくなって「これ以上大きなデータは扱えません」とお断わりのメッセージを表示するはめになるということです。

new 演算子を使えば、明らかにメモリ管理の自由度は上がります。必要に応じてメモリを確保し、必要なくなれば解放して、次に new 演算子が呼び出されたときに再利用できます。したがって、常に必要最小限のメモリしか使わずに済みます。また、コンパイル時にはデータサイズを決めておく必要はないので、どんなに大きなデータであっても対処可能です。グローバル変数やローカル変数のように、コンパイル時に確保されるメモリ領域が決定されるメモリ管理は「静的」と表現され、new 演算子のように、実行時に使用するメモリ領域が決定されるメモリ管理は「動的」と表現されます。

このように便利な動的なメモリ管理を利用するために、ポインタは欠かせない道具となります。

なお、関数の戻り値として、関数内のローカル変数へのポインタを返すことがないように注意してください。上述したように、関数内のローカル変数は、関数の終了とともに削除されますから、関数の戻り値として返したポインタは、関数の終了後にはすでに無意味なメモリ領域を指すことになってしまいます。

B.4　ポインタの使い方

最後に、ポインタにまつわる細かな話題をピックアップしてまとめます。

● NULLポインタ

初期化されていないポインタや、特別な状態にあるポインタを表現するための値として、NULL 値が定義されています。たとえば、次のようにして使います。

```
char* p = NULL;

if (p == NULL)
  p = new char[10];
```

Visual C++では、NULL 値は「(void*) 0」として定義されています。アドレス 0x0 はシステムによって保護されているので、アクセスするとシステムによってプログラムは強制終了されます。こうしたことが起こるプログラムにはバグがあるわけですが、プログラムが強制終了されれば、どこで問題が発生したのかを突き止めやすく、デバッグも比較的楽になります。もしポインタの値が不正な値だった場合、あたかも正常にメモリをアクセスしたかのようにプログラムが続行されてしまうかもしれません。そうなるとデバッグは非常に困難になります。

したがって、ポインタがまだ初期化されていないときや、メモリを解放してポインタが参照しているメモリ領域が使えなくなったときには、ポインタの値を NULL にしておく

と、面倒なトラブルを回避することができます。

●ポインタの演算

ポインタに格納されるデータはアドレスという数値ではありますが、特殊な数値であるため、ポインタに対して行える演算は加減算だけに制限されています。ポインタの値を増減させて、連続したメモリ領域をアクセスすることはあっても、ポインタの値を2倍したアドレスをアクセスするなどということは、現実としてありえません。そこで、C++言語では、最初から乗除算はできないようになっているのです。

ところで、ポインタに対する加減算には、通常の数値への演算と同じように、+、-、++、--が利用できますが、その動作は少し特殊です。

```
int* p = <初期値>;
p = p + 1;
```

以上のコードを実行したときに、+演算子で増加するポインタpの値は「1」ではありません。正解は「sizeof(int) = 4」です（Pentiumプロセッサの場合）。すなわち、ポインタ変数が指し示している変数型のサイズだけ増加するのです。考えてみれば、非常に合理的な仕様であることがわかるでしょう。たとえば、int型配列をint*型ポインタでアクセスしているとしましょう。配列の要素数が10であれば、sizeof(int) * 10 = 40バイトの領域が確保され、sizeof(int) = 4バイトずつ隙間なく各要素が並べられているわけです。これをポインタで順にアクセスするためには、ポインタの値を4バイトずつずらしていかなければなりません。このため、ポインタに対する+演算子はint型変数やchar型変数と違って、特殊な動作をするようになっているのです。

もちろん、-演算子や--演算子で減少するポインタ値、それに++演算子も同様です。ポインタに対する演算は、単位はバイトではなく、変数型のサイズを1単位とする特殊な単位となります。

もう1つ、++、--演算子に関連する注意点としては、次の表現があります。

```
*p++
```

メモリ領域を順に走査する場合によく用いられる表現ですが、この++演算子で増加するのはポインタ変数pのアドレス値であって、*pでアクセスされるメモリの内容ではありません。これは*演算子（間接演算子）よりも++演算子の方が優先順位が高いためです。つまり、

```
*(p++)
```

と解釈されます。もしメモリの内容を増加させたければ、

```
(*p)++
```

としなければなりません。

typedefされたポインタ型

Visual C++の開発環境では、たくさんのクラスが定義されていますが、同じようにtypedef指定子で定義された型が数多く存在します。そして、これらの中には、ポインタを含む型も多く含まれています。たとえば、LPCTSTR型は次のように定義されています。

```
typedef const char far* LPCTSTR;
```

このように型の定義部にアスタリスクが含まれているため、

```
LPCTSTR p;
```

のように、アスタリスクを含まずに変数を定義しても、この変数pはポインタとなります。このため混乱するかもしれませんが、ポインタ型として定義されている型は、必ずPまたはLPで始まる名前が付けられていることさえ覚えておけば、問題はありません。

この前置詞的に使われているPはいうまでもなくPointerのPです。そしてLPはLong PointerのLPです。なぜPとLPに分かれているのか、そしてLong Pointerとは何か、ということは現在では気にする必要はありません。Windows 3.1以前のWindowsシステムでのみ意味のある違いであり、Windows 95以降/Windows NTでは、PとLPは同じと考えて差し支えありません。

C 非ドキュメントービュー・アーキテクチャアプリケーションの作成

本書の第3部以降で解説したように、ドキュメントービュー・アーキテクチャはMFCの根幹を成す仕組みです。ドキュメントクラスとビュークラスを使わずには実現の難しい、さまざまな機能（データファイルの入出力、印刷など）を提供してくれるため、もはやドキュメントービュー・アーキテクチャなしにはWindowsアプリケーションを開発できなくなっているかもしれません。

しかし、第3部より前では実質的にはドキュメントービュー・アーキテクチャを使わずにサンプルプログラムを作成していたことからもわかるように、MFCにとってドキュメントービュー・アーキテクチャは必須の存在ではありません。とくにデータファイルの保存と読み込みを行う必要のないアプリケーションでは、むしろ重荷にさえ感じられるかもしれません。また、ビューが1つだけであれば（ほとんどのアプリケーションは1つで十分でしょう）、ドキュメントクラスとビュークラスにわかれているのは手間がかかるだけで、メリットはそれほど多くはありません（拡張性を考えればメリットはありますが）。

つまり、ドキュメントービュー・アーキテクチャは高機能なサービスを提供するものの、作成するアプリケーションに適していなければ、宝の持ち腐れどころか、開発の足をひっぱることになりかねないということです。

Visual C++ 5.0では、AppWizardでダイアログベースのアプリケーションタイプを指定すれば、ドキュメントービュー・アーキテクチャを使わないMFCアプリケーションを作成できましたが、ダイアログベースでなければならない点に不満が残るものでした。AppWizardを一切使わずに1からコーディングを始めればダイアログベースでなく、またドキュメントービュー・アーキテクチャを使わないアプリケーションを作成することはできましたが、難易度があまりに高いという問題があります。さらにClassWizardのような周辺ツールを活用するには、メッセージマップのように本来の開発作業とは関係ない部分まで手作業で作らなければならず、手間がかかってしまいました。

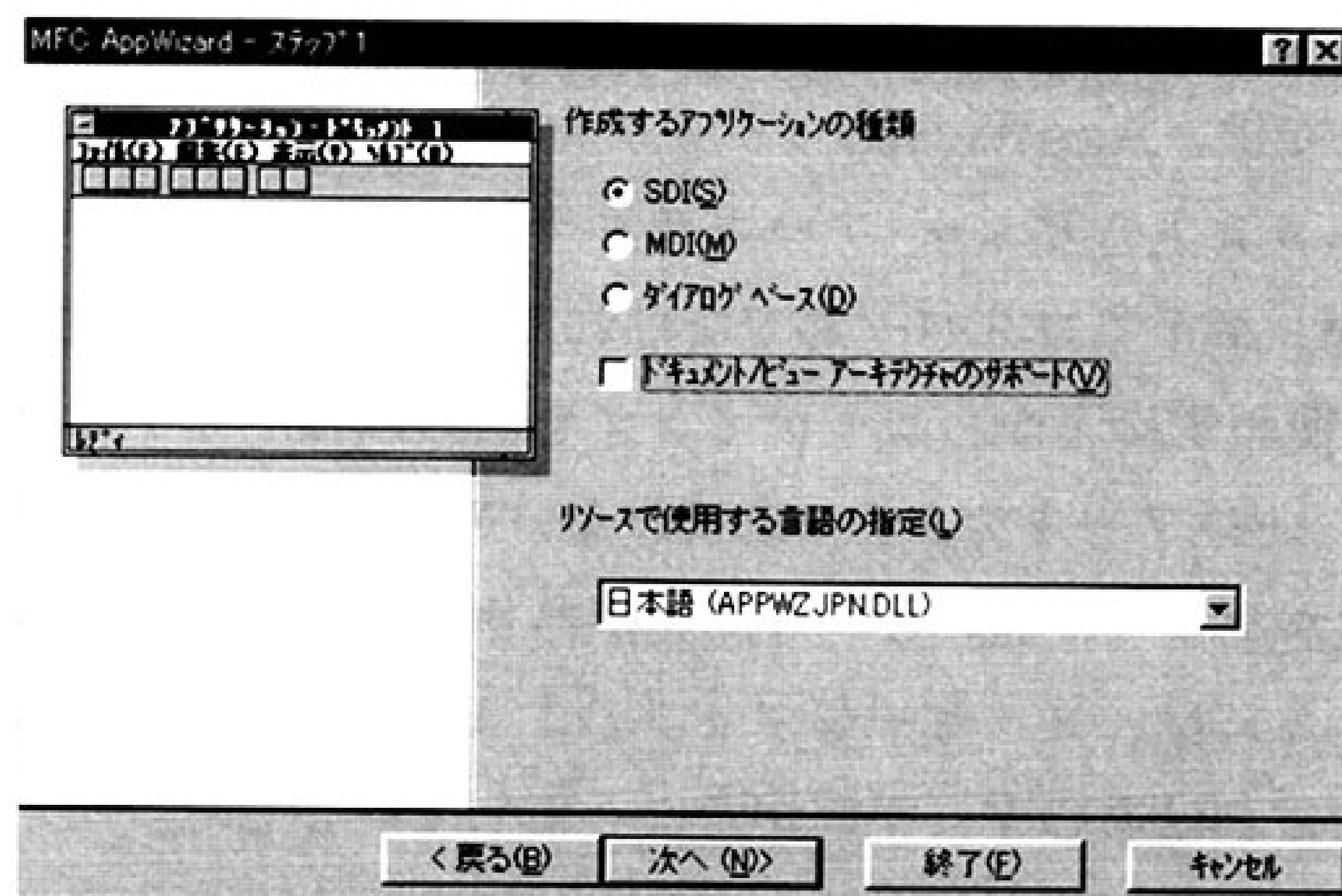
しかし、Visual C++ 6.0になって、こうしたジレンマは解消されました。AppWizardが拡張され、SDIやMDIアプリケーションタイプでも、ドキュメントービュー・アーキテクチャを使わないスケルトンを生成できるようになったのです。ドキュメントクラスもビュークラスも使わないのですから、数々の制約はありますが、スケルトンはすっきりして見通しのよいものになりますし、通常どおりClassWizardを利用してメッセージハンドラを作成することもできます。Visual C++はいっそう幅広いスタイルのアプリケーション作成に利用できる体制が整ったといえるでしょう。

C.1 ドキュメントービュー・アーキテクチャを使わないMFCアプリケーションの作り方

それでは、ドキュメントービュー・アーキテクチャを使わずにMFCアプリケーションを開発すると、どのような制約があるのでしょうか。これを知るためには、何はともあれ、AppWizardを起動して、この新しいスタイルのスケルトンを生成してみることです。ここでは本格的なアプリケーション開発までは立ち入りませんが、これまでどおり、操作手順を追う形で解説していくことにします。

いつものように、メニューから[ファイル]−[新規作成]を実行して、新規プロジェクトを作成してください。[新規作成]ダイアログボックスでは「MFC AppWizard(EXE)」を選択し、[プロジェクト名]に「NoDocView」と入力してください。

ここで[OK]ボタンをクリックすると6ステップからなるAppWizardによるスケルトンのカスタマイズ設定が始まるわけですが、ステップ1にある[ドキュメント/ビューアーキテクチャのサポート]のチェックをはずすと、ドキュメントービュー・アーキテクチャを使わないスケルトンコードが生成されるようになります。



図C-1 ドキュメント/ビューアーキテクチャのサポートのチェックをはずす

ここではシンプルに話を進めるために、［作成するアプリケーションの種類］で［SDI］を選択してください。これ以降はすべてデフォルトのまま最後までAppWizardを進めます。

［次へ］ボタンをクリックすると残り5ステップが続きますが、［ドキュメント／ビューアーキテクチャのサポート］のチェックを外したことによって、いくつかの選択項目がグレイになり、選択できなくなります。これがドキュメントービュー・アーキテクチャを使わない代償というわけです。これを見ていくと、ある程度使った場合との違いがはっきりしてきます。

まずステップ2のデータベースサポートを見ると、データベースビューを使う項目が選択できなくなっています。データベースは本書の範囲を超えるので扱っていませんが、データベースビューは名前のとおり、ビュークラスの派生クラスを利用するサービスです。したがって、これは利用できません。なお、データベースビューに限らず、CViewクラスを含めて、CScrollViewクラスやCFormViewクラスなど、CViewクラスの派生クラスは一切利用できません。

次にステップ3の複合ドキュメントサポートを見ると、複合ドキュメントのサービスはいっさい使えないことがわかります。複合ドキュメントが何かはともかく、こちらはドキュメントクラスを基本にしたサービスであるため、やはり利用できません。

このあとのステップでは、ビュークラスで提供されている印刷サービスが使えなくなっていることと、Windowsエクスプローラスタイルが選択できなくなっていることがわかります。

以上のように、ドキュメントクラスとビュークラスのどちらかに依存しているサービスはまったく使えません。

逆に、メインフレームウィンドウで管理されるツールバーやステータスバーについては通常どおり利用できるので、これらのサービスが欠かせない場合でも問題にはなりません。

C.2 作成されるクラス

AppWizardの最後になるステップ6を見れば、ドキュメントクラスもビュークラスもないことがはっきりとわかります。



図C-2　ステップ6：AppWizardで作成される新規アプリケーションクラス

AppWizardによって作成されるのは、**表C-1**に示すたった3つのクラスだけです。また、これらの基底クラスは変更することはできません。

CNoDocViewApp	CWinAppクラスの派生クラス
CMainFrame	CFrameWndクラスの派生クラス
CChildView	CWndクラスの派生クラス

表C-1　作成されるクラス

ここで注意してほしいのは、CChildViewクラスの基底クラスはCViewクラスではなく、CWndクラスであるということです。名前だけ見ると勘違いしそうですが、CWndクラスという、すべてのウィンドウ系クラスの基底クラスとなっている、非常にプリミティブなクラスです。しかし、Viewの付く名前を持つだけあって、CChildViewクラスの用途はビュークラスによく似ています。**図C-3**に示すように、CMainFrameクラスが管理するメインフレームウィンドウが外側にあり、そのクライアント領域を埋め尽くすように、CChildViewクラスが管理するウィンドウが納まります。これはドキュメントービュー・アーキテクチャを使ったときのビュークラスにそっくりです。もっともCWndクラスはCViewクラスの基底クラスですから、CViewクラスに比べてサービスが貧弱であることは明白です。この点の違いについては、後ほど解説します。

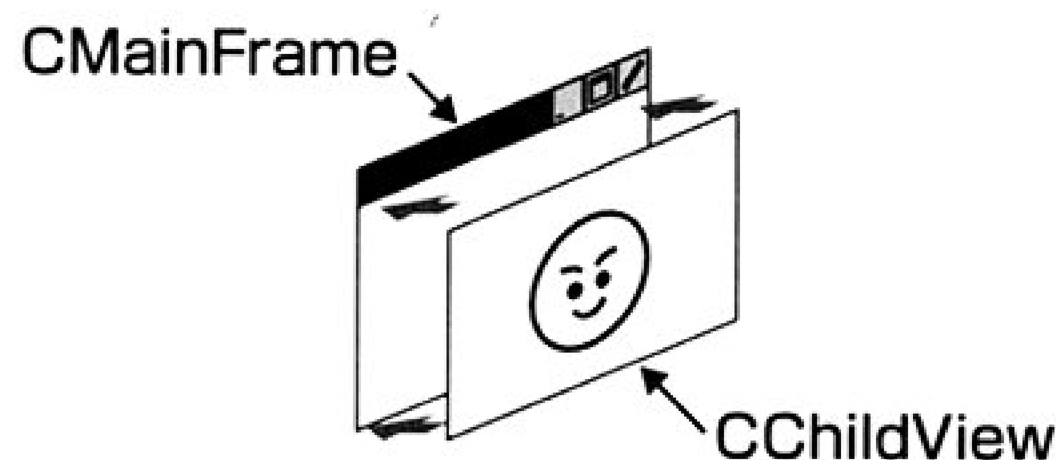


図 C-3　CMainFrame クラスと CChildView クラスの関係

C.3 ウィンドウを開いて、閉じるだけのスケルトン

こうして作成されたプロジェクトをビルドすれば、さらに違いは鮮明になります。

図 C-4 に実行した NoDocView を示します。AppWizard で生成した直後にビルドしただけですから、何もできないのは当然ですが、予想以上に何もできません。

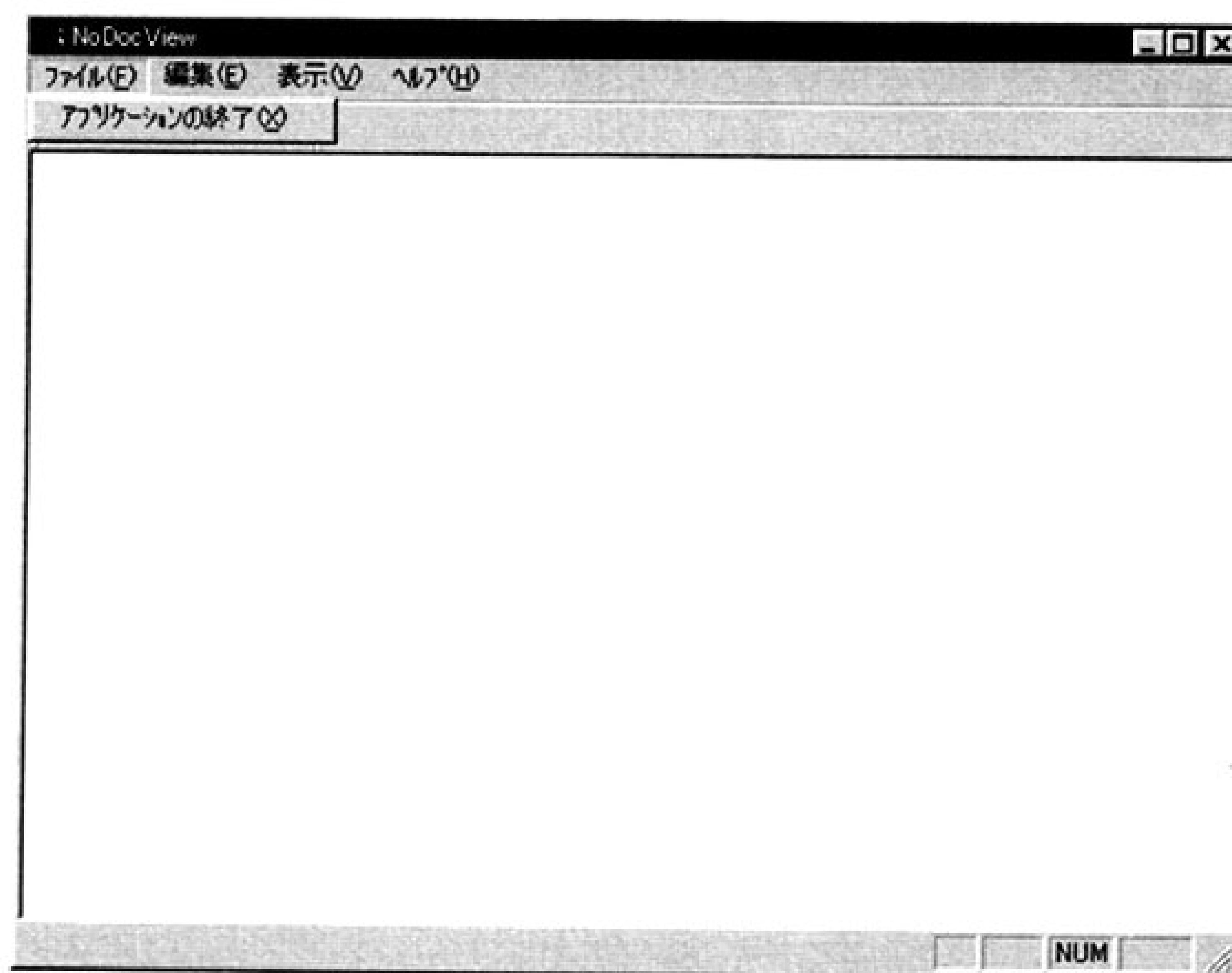


図 C-4　NoDocView の実行画面

図 C-4 で示したように、［ファイル］メニューには［アプリケーションの終了］以外には何もありません。ドキュメントクラスがないので［新規作成］はありませんし、［開く］や［保存］もありません。つまり、ファイルオープンダイアログを開くことすらできないのです。［編集］メニューには［コピー］や［切り取り］など標準的なコマンドが用意されていますが、これらメニューコマンドのハンドラが存在しないのでグレイ表示されるだけで何もできません。もっとも、ドキュメント－ビュー・アーキテクチャを使ったスケルトンでも、この点については同じでしたが。かろうじて、［表示］メニューでツールバーとステータスバーの表示／非表示を切り替えたり、［ヘルプ］メニューからアバウトボックスの表示がで

きるにすぎません。

つまり、作成されたスケルトンはウィンドウを表示し、閉じることしかできないのです。

C.4 スケルトンの内容

このように何もできないスケルトンですから、相応にコードサイズも小さなものです。ここでは簡単にドキュメントービュー・アーキテクチャを使った場合と比較して、上述した3つのクラスに用意されている主要なメンバを解説します。

- **CNoDocViewApp::InitInstance 仮想メンバ関数**

  CWinApp::InitInstance 仮想メンバ関数はアプリケーションが実行された直後に初期化処理を行う場所です。ドキュメントービュー・アーキテクチャを使った場合は、ここでドキュメントクラスとビュークラス、そしてフレームウィンドウクラスを結び付けるために、ドキュメントテンプレートが定義されていましたが、もちろんこの処理はなくなっています。

  また、コマンドラインから起動されたときにパラメータを解析して、ファイルを開いたりする処理も省略されています。扱うべきドキュメントクラスがないのですから、当然の措置と言えるでしょう。

- **CMainFrame::m_wndView メンバ変数**

  ドキュメントービュー・アーキテクチャを使う場合は、ビュークラスはドキュメントクラスの管理下にあったため、CView オブジェクトは CDocument クラスのメンバ変数として管理されていました。それに対して使わない場合は、メインフレームウィンドウを管理する CMainFrame クラス型のメンバ変数として、ビュークラスに相当する CChildView オブジェクトが定義されています。それが CMainFrame::m_wndView メンバ変数です。

- **CMainFrame::OnCreate 仮想メンバ関数**

  上述した CMainFrame::m_wndView メンバ変数を使って、実際に CChildView クラスに対応するウィンドウを作成する箇所が CMainFrame::OnCreate 仮想メンバ関数です。CChildView クラスに対応するウィンドウは、メインフレームウィンドウの子ウィンドウとして作成されるため、メインフレームウィンドウ（親ウィンドウ）が作成されたあとに作成する必要があります（親がなければ子は産めません）。そこで、ウィンドウが作成された直後に呼び出される OnCreate 仮想メンバ関数が使われています。この仮想メンバ関数ではほかにも、ツールバーオブジェクトやステータスバーオブジェクトの作成が同時に行われますが、これはドキュメントービュー・アーキテクチャを使った場合と同じです。

● **CChildView::OnPaint 仮想メンバ関数**

ドキュメントービュー・アーキテクチャを使った場合には、画面表示は常に CView::OnDraw 仮想メンバ関数で処理していました。しかしビュークラスはありませんから、代替クラスである CChildView クラスの OnPaint 仮想メンバ関数がこれに代わって使われます。

CWnd::OnPaint 仮想メンバ関数は WM_PAINT メッセージのメッセージハンドラとしてあらかじめ定義されている関数です。実は CView クラスでも CView::OnPaint 仮想メンバ関数から CView::OnDraw 仮想メンバ関数を呼び出しているだけだったので、OnDraw 仮想メンバ関数と OnPaint 仮想メンバ関数はほぼ同じものと考えることができます。ただし、CView クラスでは、画面表示と印刷の両方とも CView::OnDraw 仮想メンバ関数で行うという目的があったため、CView::OnPaint 仮想メンバ関数と CView::OnDraw メンバ関数は分けられていたのです。しかし、印刷をサポートしない CChildView クラスでは意味のない手間なので、直接 OnPaint 仮想メンバ関数に表示処理を記述します。

C.5 スケルトンへコードを追加する

さて、こうして作成されたスケルトンに手を加える段階に来たとき、違いはあるのでしょうか。ドキュメントービュー・アーキテクチャによって提供されていた多くのサービスが削除されているのですから、仕事が増えるのはもちろんですが、嬉しいことに開発環境の支援ツールはそのまま利用できます。ドキュメントービュー・アーキテクチャを使っていないとはいえ、作成されたスケルトンが MFC をベースとしていることは変わりません。そして開発支援ツールは MFC を使ってさえいれば、問題なく動作します。したがって、ClassWizard を使ってメッセージハンドラや仮想メンバ関数のオーバーライドが行えます。またリソースエディタを使って、メニューやダイアログボックスをエディットできます。したがって、スケルトンの動作さえ把握できていれば、何の問題もなく、これまでどおりのコーディングを行えます。

ただし、1 つだけ問題があります。Visual C++ 6.0 の初期バージョンでは AppWizard に問題があり、ドキュメントービュー・アーキテクチャを使わずにスケルトンを作成した場合、ClassWizard を開いても、正常にハンドルできるメッセージが一覧表示されません。これはクラスごとに設定されているメッセージフィルタ（チャイルドウィンドウやフレームウィンドウのように、クラスが管理しているウィンドウの性格に対応して、ハンドルできるメッセージに制限を加える仕組み）が正常にセットアップされないためです。この問題を解決するためには、ClassWizard ではなく、WizardBar から［Windows メッセージ

ハンドラの追加]を実行します。するとClassWizardよりも簡素化されたメッセージハンドラの定義ダイアログボックスが表示されます。このダイアログボックスには[クラスで使用可能なメッセージ用フィルタ]があり、ここで正常なメッセージフィルタを設定することによって、この問題を解決できます。CChildViewクラスには[チャイルドウィンドウ]を、CMainFrameクラスには[トップモーストウィンドウ]を指定すれば、その後はClassWizardでも正常にメッセージが一覧されるようになります。

索 引

Symbols

A

B

C

D

W

ア

イ

ウ

エ

オ

カ

キ

ク

ケ

コ

サ

シ

ス

セ

ソ

タ

Visual C++6.0 プログラミング入門

著　者　桜田 幸嗣／田口 景介
発行所　**株式会社アスキー**
〒151-8024　東京都渋谷区代々木4-33-10

Printed in Singapore

● 1477701

Visual C++6.0
プログラミング入門

X04－62610

disc

254-CD-002

ASCII

パーソナルプログラミングシリーズ

Visual C++6.0 プログラミング入門

●対象機種および環境

Visual C++ 6.0が動作する機種および環境

●フロッピーディスクの内容

本書に掲載されているサンプルプログラムのソースコード

●注意

CD-ROMには実行可能ファイルは含まれていません。また、ソースコードのコンパイルにはVisual C++ 6.0が必要です。

CD-ROMに収められているソースコードを利用する前に、ハードディスクにそれらのファイルをすべてコピーして、読み取り専用属性をはずす作業が必要です。詳しくは本書の「付属CD-ROMについて」を参照してください。

CD-ROMには、Visual C++ 6.0本体は含まれていません。別途ご用意ください。

第1部 Visual C++に触ってみよう

1章 Visual C++とは？
2章 Visual C++流プログラミング!!
3章 プログラムを解剖してみよう
4章 フレームワークを解剖してみよう

第2部 Visual C++プログラミングの基本を押さえよう

1章 GDIはグラフィックス表示の合言葉
2章 メニューを使ってみよう
3章 ダイアログボックスを使ってみよう
4章 マウスとキーボードからの入力
5章 デバッグしてみよう

第3部 MFCを使ってみよう

1章 テキストエディタを作ってみよう
2章 ドローツールを作ってみよう

第4部 Windowsらしいアプリケーションを作ってみよう

1章 URLマネージャの概要
2章 プログラム作成の前準備
3章 ツリービューコントロールを使ってみよう
4章 もう少しWWWブラウザらしく



254-600-901